Latency-driven replication for globally distributed systems

Szymaniak, M.P.

2007

document version
Publisher's PDF, also known as Version of record

LATENCY-DRIVEN REPLICATION FOR GLOBALLY DISTRIBUTED SYSTEMS

MICHAŁ SZYMANIAK

VRIJE UNIVERSITEIT

LATENCY-DRIVEN REPLICATION FOR GLOBALLY DISTRIBUTED SYSTEMS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 17 april 2007 om 10.45 uur
in de aula van de universiteit,
De Boelelaan 1105

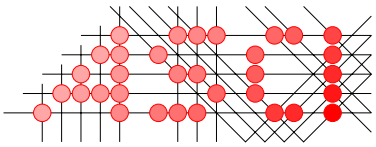
door

MICHAŁ PIOTR SZYMANIAK

geboren te Lublin, Polen

promotor: prof.dr.ir. M.R. van Steen
copromotor: dr.ir. G.E.O. Pierre

To my parents, Jadwiga and Gabriel,
and to my wife Ewa



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 141.

Parts of Chapter 2 have been published in the *ACM Computing Surveys*.

Parts of Chapter 3 have been published in the *Proceedings of the 10th IEEE International Conference on Parallel and Distributed Systems*.

Parts of Chapter 4 have been published in the *2005 IEEE International Symposium on Applications and the Internet*, and in the *IPSJ Journal*.

Parts of Chapter 5 have been published in the *Proceedings of the ICST International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Many people helped me during my PhD studies, both when I was doing my research and when writing this very dissertation. There is no doubt that it would not have its current shape without all the support I received from various people around me, and so it is now the time to properly thank them all.

The truth is I did not even think about pursuing a PhD before I met my advisors, Maarten van Steen and Guillaume Pierre. Their broad academic experience combined with relaxed style and infectious enthusiasm for distributed systems turned the process of writing my master's thesis into an exciting intellectual adventure. It was this exceptional atmosphere of working together that convinced me to start PhD studies under their supervision. As expected, the adventure continued also during the PhD stage. Both Maarten and Guillaume proved to be excellent tutors, patiently explaining the rules of scientific research, even though the outcome was seldom immediately visible. They always had time for brainstorming sessions and carefully corrected all the drafts of publications-to-be, regardless of how.. premature some of them were. I believe that the most stimulating were Maarten and Guillaume's enthusiastic reactions to my findings, followed by fountains of what-ifs to be investigated next. The ability to identify such what-ifs is what lies at the core of research, and it is definitely the rare talent of Maarten and Guillaume that they not only use that ability, but also foster it in their students.

Apart from working with Guillaume at the university, I also had the pleasure to be his guest at informal meetings regularly organized by him and his wife Caroline. I was genuinely impressed by their hospitality and cooking talent, and truly appreciate that they made me feel more at home abroad.

Getting back on the scientific ground, I would like to thank all the members of my reading committee: Michele Colajanni, Pablo Rodriguez Rodriguez, Piet van Mieghem, Herbert Bos, and Hans Akkermans. All of them spent a lot of time and effort reviewing this dissertation, and their valuable comments allowed me to make many last-minute improvements.

Long before this dissertation was complete, several people helped me during my work on particular topics. David Presotto, my supervisor during my internship at Google, arranged for everything needed to develop the large-scale latency estimation system. Mariana Simons-Nikolova, a researcher at Philips Research Labs, verified the concept of distributed handoffs from the perspective of their compatibility with IPv6. Henk Uiterwaal, a senior project manager at RIPE, provided real-world traces for various experiments with latency estimation and replica placement. Andrew Tanenbaum, the head of the Computer Systems group at the Vrije Universiteit, let me instrument his popular Web site on US elections with my scripts collecting latency information.

And so we have come to my colleagues here at the VU. The last five years would not have been the same without them, especially without my office-mates: Swami and Spyros. We were extremely lucky to get on so well, which made all these years spent in the same office very pleasant. Both Swami and Spyros have already mentioned in the prefaces to their respective dissertations that the three of us spent an enormous amount of time discussing all kinds of things. I can only say that I, too, enjoyed our discussions a lot.

Still speaking about my colleagues from the university, a very special thanks must go to Arno, who translated the dissertation summary into the Dutch 'samenvatting' and helped me to prepare this entire book for printing. I would also like to thank Berry, Bogdan, Daniela, Guido, Jan-Mark, Konrad, Paolo, Srijith, and Willem for bringing fun to our workplace.

Then it is time for all my friends from Amsterdam. My two best friends, Konrad and Marcin, moved with me to the Netherlands and will soon be writing their own dissertations. I am very happy that we were able to support one another throughout all these years, and that we succeeded in finishing our PhDs as planned. Gosia and Robert, my two friends from the one-year masters program, also did their PhD in Amsterdam and were my flat-mates for the first two years. I enjoyed that time a lot. Finally, all the Polish people that I shared my time with during numerous parties, trips, and cultural events. They kept me away from loneliness: Asia, Wojtek, Kasia, Paweł, Wojtek, Marta, Ania, Łukasz, Radek, Magda, Ania, Ilona, Bartosz, Paweł, Ula, Michał, and Teresa. Thank you all.

I could never write this dissertation without the continuous support from my family. My parents were always there when I needed them, and I could always count on their advice and understanding. My brother Bartosz and my sister Kasia with her husband Radek made sure that I had a great time during all the long-awaited trips to our home in Poland. I am deeply grateful to all of them for this special feeling of being loved that has always been with me regardless of the physical distance between us.

Finally, Amsterdam shall always be a very special place for me, as this is where I met my wife Ewa. Her love and trust have been my driving force since we began our journey together, and I hope that one day I will be able to thank her enough for that.

Michał Szymaniak
Amsterdam, March 2007

CHAPTER 1

Introduction

Over the last two decades, the Internet has evolved from an experimental research network into a global communication medium. It is now being used on a daily basis by millions of people, who send email messages, read news on the Web, or shop on thousands of e-commerce sites.

Without doubt, the Internet streamlines many of everyday activities. For example, Internet banking enables its users to take care of their finances without forcing them to wait in queues or to punch account numbers on a telephone dial. Similarly, Internet shopping allows for acquiring goods without leaving home and without reciting the shopping list over the phone. Likewise, Internet ticket reservation systems are not only very convenient to use, but they also enable many airlines to save on their maintenance costs by considerably reducing staff at their call centers, causing the flying tickets themselves to become cheaper.

What is more important, however, is that the Internet provides completely new opportunities for its users. For example, Internet users have access to information published literally anywhere in the world. People playing computer games, in turn, can compete with adversaries from thousands of miles apart using interactive gaming sites. Also, both individuals and entire communities can express themselves and be heard by millions of people reading blogs. All these advantages of the Internet contribute to the continuous increase in its popularity, causing the global network to span an increasingly wider geographical area and to connect more and more machines.

The growth of the Internet poses new challenges to the architects of Internet services. In order to be successful, an Internet service must remain convenient to use despite any foreseeable increase in both network size and user population. From a human perspective, convenience may mean many different properties, which can essentially be split into two groups. The first group are the properties of information returned by the service, such as the easiness of navigation through that

information, or its availability in multiple languages. These aspects are generally investigated by studies in human-computer interaction [Shneiderman, 2004].

The second group of service properties that affect usage convenience consists of those related to the quality of communication with the service. These properties include, for example, service availability, communication security, and service response time to user requests. Whereas individual judgments on the returned information quality are by nature subjective, all users generally expect Internet services to be available all the time, accessible without risk, and responding as fast as possible [Shneiderman, 1984]. For example, news Web sites should deliver their articles whenever requested, Internet shops should provide secure payment mechanisms, and Web search engines should return their search results immediately. Not surprisingly, it has been shown that users prefer safe and reliable Internet services featuring short response times, and that faulty, suspicious, or slow services tend to quickly lose their popularity [Nielsen, 1999]. This motivates the research into improving the quality of communication with Internet services. In particular, the research question addressed by this thesis is: *how to improve the response times of large-scale Internet services?* As for the remaining two factors affecting communication quality, namely service availability and communication security, we assume them to be achieved by means of existing techniques. Some of these techniques are mentioned in Chapter 2, where we discuss related research efforts.

The response time of an Internet service consists of two parts. First, when it receives a request, the service needs some time to *generate* its response. Second, some more time is needed to *transmit* the generated response to the user machine. The techniques presented in this thesis aim at reducing the transmission delay, and are independent of those used for response generation. As a consequence, our techniques treat all the response data as if they have been generated in advance, which is equivalent to assuming that all the replicated data are static. Note that this assumption does not prevent our proposed techniques from being used in systems generating responses on-demand, as such systems can always combine our solutions with their own techniques for dynamic response generation. We briefly discuss a number of such techniques in Chapter 2.

Transmission delays essentially depend on three factors [Tanenbaum, 2003]. The first factor is the bandwidth available in the network path between an Internet service and a given user machine, which determines the rate at which the Internet service sends its responses to that user machine. The second factor is the latency between the Internet service and the user machine, which is simply the time needed to transfer the smallest piece of data between the communicating parties. The third factor is the packet loss rate, which essentially defines how much data on average needs to be retransmitted in order to ensure that all the data have been transferred successfully.

A crucial observation at this point is that the above three factors are not equally relevant when optimizing the transmission delay. For example, packet loss is likely to occur on congested network paths, where intermediate routers lack the bandwidth to send all the packets and therefore must drop some of them. However, typical packet loss rates are lower than 1% on backbone networks [Iannaccone et al., 2002], which exploit high-quality infrastructures and overprovision available bandwidth to avoid congestion [Fraleigh et al., 2003]. On the other hand, these loss rates are likely to grow on slow or unreliable links next to user machines, such as DSL lines or wireless networks. Nevertheless, higher packet loss rates in user-side networks are impossible to tackle by Internet services having no control over such networks.

From the remaining two factors, bandwidth intuitively seems to be more important than latency, as it determines the ultimate data transmission rate. However, a study in Internet traffic dynamics shows that bandwidth available to Internet services is often limited not by network capacity, but by network latency [Allman, 2000]. Such a phenomenon can be observed when an Internet service communicates with user machines using TCP connections, as the throughput of a TCP connection is constrained by the round-trip time (*rtt*) between the communicating parties. The *rtt* determines the time necessary to acknowledge the transmitted data, and the amount of unacknowledged data sent over a TCP connection cannot exceed the fixed size B_{recv} of the TCP buffer on the receiving machine. The throughput of a single TCP connection T_{conn} is therefore limited by:

$$T_{conn} = \frac{B_{recv}}{rtt}$$

The above limitation can be observed even when user machines communicate with Internet services over networks with nearly infinite bandwidth. As a consequence, simply overprovisioning the available bandwidth alone might turn out to be insufficient to improve the transmission delay, unless it is accompanied by simultaneous optimization of latencies between the Internet service and the user machines.

This thesis addresses the problem of latency optimization. The fundamental difficulty here is that the latencies of (non-congested) network paths are determined by physical properties of links forming these paths, such as the speed of light in a given physical medium. Given that changing these properties is impossible, the only way to reduce latencies is to reorganize the communication with the Internet service such that the service responses are transmitted over short network paths.

However, when the user population is scattered over the entire world, communicating with every user machine over a short network path is impossible. Large-scale Internet services often solve this problem by means of replication, in which

the service is deployed in many instances located in different parts of the Internet [Brewer, 2001]. Each instance can then handle requests from user machines in its proximity, which allows for using short network paths for communication. This results in low network latencies between the service and its clients, which reduces the service response times and hence makes the service more attractive for its users.

Replication has been commonly employed by many different kinds of Internet services, especially those managing large collections of data. Examples include content delivery networks [Dilley et al., 2002; Gao et al., 2003; Pierre and van Steen, 2006], peer-to-peer file sharing platforms [Saroiu et al., 2002; Cohen and Shenker, 2002], and databases [Bernstein and Goodman, 1983; Petersen et al., 1996]. However, most of the recent research into global-scale replication has been done in the context of the Web. We therefore also focus on the Web environment when proposing our solutions, although many of them are applicable to other systems as well.

Systems implementing Web replication are commonly referred to as Web replica hosting systems [Sivasubramanian et al., 2004]. Each such a system consists of a number of replica servers, which host replicas of Web documents served by the system. Whenever a Web client wishes to download any such document, the system informs the Web client about a replica server nearby at which a replica of that document can be found. The Web client can then contact the replica server and retrieve the replica contents. As we focus our discussion on *transmission delays*, we assume these contents to be static. We therefore do not deal with issues related to dynamic content generation, even though our solutions could also be incorporated into systems generating Web content on-demand.

Although the operation of a Web replica hosting system seems relatively simple, each such system needs to address many specific issues. For example, preserving high system performance requires that it is continuously evaluated such that the system knows when to adjust its operation. In particular, *latency-driven* systems must be able to estimate latencies between their individual nodes (client and replica servers). Such latency information is necessary to make correct decisions on creating new replicas on specific replica servers, and configuring each of them to be used by a given group of clients. In a latency-driven system, both these decisions must be made using latency-aware techniques.

This thesis presents a number of techniques for latency-driven replication in *large-scale* systems. We demonstrate that several problems are particularly difficult when a system consists of millions of nodes, and propose solutions crafted especially for such situations. These solutions concentrate on three issues that we believe affect the transmission delay most: latency modeling, replica placement, and client redirection.

Estimating inter-node latencies in a large-scale system is difficult, because there are a tremendous number of node pairs. Given that latencies tend to fluctuate continuously, repetitively measuring latencies between all the node pairs is impractical. Instead, the system needs to employ techniques that provide large numbers of latency estimates at low cost. Our solutions exploit the concept of network positioning, which models the Internet as a multi-dimensional geometric space [Ng and Zhang, 2002]. In that concept, the system associates each node with its position in the geometric space and estimates latencies between nodes as Euclidean distances between their corresponding positions. The low number of measurements necessary for node positioning and the purely numerical nature of latency estimation make this approach specifically attractive for large-scale systems. Chapter 3 discusses how to efficiently implement network positioning in both centralized and federated environments, and presents several experiments based on real-life traces to prove the high accuracy of the obtained latency estimates.

System models based on network positioning can be used to decide on replica placement. However, the problem is that each replica is typically accessed by a large number of clients scattered over multiple parts of the Internet. Identifying the replica locations that minimize network latencies for all the clients is not trivial, and so far required time-consuming computations during which massive numbers of pair-wise latencies were estimated [Karlsson et al., 2002]. Such computations are impractical in a real-life system that decides on the placement of many replicas. We therefore propose a novel replica placement algorithm that exploits the geometric properties of models produced with network positioning. It identifies the best replica locations as the clusters of node positions in the geometric space, and carefully maps replicas to clusters to avoid overlaps. Chapter 4 describes the details of our algorithm, and demonstrates that it generates high-quality replica placements several orders of magnitude faster than previous solutions.

Clusters chosen by our replica placement algorithm correspond to network regions rather than individual machines. As a consequence, another set of techniques must be used to map replicas to individual replica servers within network regions. However, given that the properties of our latency models guarantee that all the replica servers within a given region are proximal to one another in terms of latency, these techniques can focus on optimizing other metrics. We chose these metrics to be network bandwidth and server availability, and propose to improve them by organizing replica servers in each region into a distributed server. Such a server can offer more bandwidth by aggregating the network connection capacities of individual replica servers, and is able to retain a stable network address despite failures of individual replica servers. This allows each distributed server to become a self-organizing platform that turns a group of potentially unreliable replica

servers into a reliable hosting facility for Internet services. Chapter 5 presents our techniques enabling distributed servers to implement their stable addresses, and allowing for bandwidth aggregation by means of transparent wide-area client handoffs.

Whereas intra-region redirection is handled by the distributed servers, its inter-region counterpart needs to be performed by means of other techniques. One of the problems that these techniques need to address is how to choose the distributed server for each client such that the pair-wise latency between these two is minimized. We propose to solve this problem based on our latency models, which enable one to estimate latencies between the client and all the distributed servers, and then to identify the best of them. Another problem is how to inform each client about the redirecting decision. This can be achieved by means of DNS, which is commonly employed by large-scale systems for that purpose [Cardellini et al., 2003]. We discuss how to implement global client redirection by combining wide-area handoffs with mechanisms for domain name resolution at the end of Chapter 5.

We believe that our proposed latency-driven techniques are applicable in a variety of large-scale distributed systems. Despite being described in the context of the Web, all these techniques could also be employed by other systems that need to optimize network latencies between their components. Chapter 6 explains how each of our techniques can be perceived as a single step towards latency minimization in a globally distributed system, and proposes a number of possible directions for future research building on our results.

Latency-driven techniques described in this thesis are especially crafted for large-scale environments, where modeling and processing network latency information is exceptionally difficult. We hope that following our approaches to latency modeling, replica placement, and client redirection shall enable large-scale systems of various kind to improve their performance in terms of access- and transmission delays, and hence to increase their attractiveness to Internet users.

CHAPTER 2

Related Work

2.1. INTRODUCTION

Replication is a technique that allows to improve the quality of distributed services. In the past few years, it has been increasingly applied to Web services, notably for hosting Web sites. In such cases, replication involves creating copies of a site's Web documents, and placing these document copies at well-chosen locations. Also, various measures are taken to ensure consistency of when a replicated document is updated. Finally, effort is put into redirecting a client to a server hosting a document copy such that the client is optimally served. Replication can lead to reduced client latency and network traffic by redirecting client requests to a replica closest to that client. It can also improve the availability of the system, as the failure of one replica does not result in entire service outage.

These advantages motivate many Web content providers to offer their services using systems that use replication techniques. We refer to systems providing such hosting services as *replica hosting systems*. The design space for replica hosting systems is big and seemingly complex. In this chapter, we concentrate on organizing this design space and review several important research efforts concerning the development of Web replica hosting systems. A typical example of such a system is a Content Delivery Network (CDN) [Hull, 2002; Rabinovich and Spatscheck, 2002; Verma, 2002].

There exists a wide range of articles discussing selected aspects of Web replication. In this chapter, we provide a framework that covers the important issues that need to be addressed in the design of a Web replica hosting system. The framework is built around an *objective function* – a general method for evaluating the system performance. Using this objective function, we define the role of the different system components that address separate issues in building a replica hosting system.

The Web replica hosting systems we consider are scattered across a large geographical area, notably the Internet. When designing such a system, at least the following five issues need to be addressed:

1. How do we *select and estimate the metrics* for taking replication decisions?
2. *When* do we replicate a given Web document?
3. *Where* do we place the replicas of a given document?
4. How do we *ensure consistency* of all replicas of the same document?
5. How do we *route client requests* to appropriate replicas?

Each of these five issues is to a large extent independent from the others. Once grouped together, they address all the issues constituting a generalized framework of a Web replica hosting system. Given this framework, we compare and combine several existing research efforts, and identify problems that have not been addressed by the research community before.

Another issue that should also be addressed separately is *selecting the objects* to replicate. Object selection is directly related to the granularity of replication. In practice, whole Web sites are taken as the unit for replication, but Chen et al. [Chen et al., 2002, 2003] show that grouping Web documents can considerably improve the performance of replication schemes at relatively low costs. However, as not much work has been done in this area, we have chosen to exclude object selection from our study.

We further note that Web caching is an area closely related to replication. In caching, whenever a client requests a document for the first time, the client process or the local server handling the request will fetch a copy from the document's server. Before passing it to the client, the document is stored locally in a cache. Whenever that document is requested again, it can be fetched from the cache locally. In replication, a document's server pro-actively places copies of document at various servers, anticipating that enough clients will make use of this copy. Caching and replication thus differ only in the method of creation of copies. Hence, we perceive caching infrastructures (like, for example, Akamai [Dilley et al., 2002]) also as replica hosting systems, as document distribution is initiated by the server. For more information on traditional Web caching, see [Wang, 1999]. A survey on hierarchical and distributed Web caching can be found in [Rodriguez et al., 2001].

A complete design of a Web replica hosting system cannot restrict itself to addressing the above five issues, but should also consider other aspects. The two most important ones are *security* and *fault tolerance*. From a security perspective, Web replica hosting systems provide a solution to denial-of-service (DoS) attacks.

By simply replicating content, it becomes much more difficult to prevent access to specific Web content. On the other hand, making a Web replica hosting system secure is currently done by using the same techniques as available for Web site security. Obviously, wide-spread Web content replication poses numerous security issues, but these have not been sufficiently studied so far.

In contrast, when considering fault tolerance, we face problems that have been extensively studied in the past decades with many solutions that are now being incorporated into highly replicated systems such as those studied here. Notably the solutions for achieving high availability and fault tolerance of a single site are orthogonal to achieving higher performance and accessibility in Web replica hosting systems. These solutions have been extensively documented in the literature ([Schneider, 1990; Jalote, 1994; Pradhan, 1996; Alvisi and Marzullo, 1998; Elnozahy et al., 2002]), for which reason we do not explicitly address them in our current study.

The rest of this chapter is organized as follows. In Section 2.2 we present our framework of wide-area replica hosting systems. In Sections 2.3 to 2.7, we discuss each of the above mentioned five problems forming the framework. For each problem, we refer to some of the significant related research efforts, and show how the problem was tackled. We draw our conclusions in Section 2.8.

2.2. FRAMEWORK

The goal of a replica hosting system is to provide its clients with the best available performance while consuming as little resources as possible. For example, hosting replicas on many servers spread throughout the Internet can decrease the client end-to-end latency, but is bound to increase the operational cost of the system. Replication can also introduce costs and difficulties in maintaining consistency among replicas, but the system should always continue to meet application-specific consistency constraints. The design of a replica hosting system is the result of compromises between performance, cost, and application requirements.

2.2.1. Objective function

In a sense, we are dealing with an optimization problem, which can be modeled by means of an abstract *objective function*, F_{ideal}, whose value λ is dependent on many input parameters:

$$\lambda = F_{ideal}(p_1, p_2, p_3, \ldots, p_n)$$

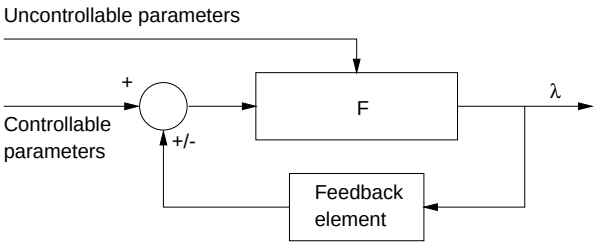


Figure 2.1: The feedback control loop for a replica hosting system.

In our case, the objective function takes two types of input parameters. The first type consists of uncontrollable system parameters, which cannot be directly controlled by the replica hosting system. Typical examples of such uncontrollable parameters are client request rates, update rates for Web documents, available network bandwidth, and network latencies between clients and replicas. The second type of input parameters are those whose value can be controlled by the system. Examples of such parameters include the number of replicas, the location of replicas, and the adopted consistency protocols.

One of the problems that replica hosting systems are confronted with, is that to achieve optimal performance, only the controllable parameters can be manipulated. As a result, continuous feedback is necessary, resulting in a traditional feedback control system as shown in Figure 2.1.

Unfortunately, the actual objective function F_{ideal} represents an ideal situation, in the sense that the function is generally only implicitly known. For example, the actual dimension of λ may be a complex combination of monetary revenues, network performance metrics, and so on. Moreover, the exact relationship between input parameters and the observed value λ may be impossible to derive. Therefore, a different approach is always followed by constructing an objective function F whose output λ is compared to an assumed optimal value λ^* of F_{ideal}. The closer λ is to λ^*, the better. In general, the system is considered to be in an *acceptable state*, if $|\lambda^* - \lambda| \leq \delta$, for some system-dependent value δ.

We perceive any large-scale Web replica hosting system to be constantly adjusting its controllable parameters to keep λ within the acceptable interval around λ^*. For example, during a flash crowd (a sudden and huge increase in the client request rate), a server's load increases, in turn increasing the time needed to service a client. These effects may result in λ falling out of the acceptable interval and that the system must adjust its controllable parameters to bring λ back to an acceptable value. The actions on controllable parameters can be such as increasing the number of replicas, or placing replicas close to the locations that generate most requests. The exact definition of the objective function F, its input parameters, the optimal value λ^*, and the value of δ are defined by the system designers

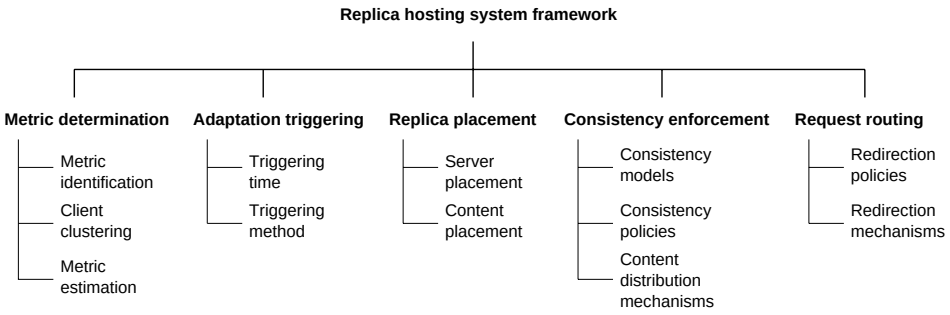


Figure 2.2: A framework for evaluating wide-area replica hosting systems.

and will generally be based on application requirements and constraints such as cost.

In this chapter, we use this notion of an objective function to describe the different components of a replica hosting system, corresponding to the different parts of the system design. These components cooperate with each other to optimize λ. They operate on the controllable parameters of the objective function, or observe its uncontrollable parameters.

2.2.2. Framework elements

We identify five main issues that have to be considered during the design of a replica hosting system: metric determination, adaptation triggering, replica placement, consistency enforcement, and request routing. These issues can be treated as chronologically ordered steps that have to be taken when transforming a centralized service into a replicated one. Our proposed framework of a replica hosting system matches these five issues as depicted in Figure 2.2. Below we discuss the five issues and show how each of them is related to the objective function.

In **metric determination**, we address the question how to find and estimate the metrics required by different components of the system. Metric determination is the problem of estimating the value of the objective function parameters. We discuss two important issues related to metric estimation that need to be addressed to build a good replica hosting system. The first issue is *metric identification*: the process of identifying the metrics that constitute the objective function the system aims to optimize. For example, a system might want to minimize network latency between the clients and the replicas, or might want to minimize the cost of replication. The other important issue is the process of *metric estimation*. This involves the design of mechanisms and services related to estimation or measurement of metrics in a scalable manner. As a concrete example, measuring client latency to

every client is generally not scalable. In this case, we need to group clients into clusters and measure client-related metrics on a per-cluster basis instead of on a per-client basis (we call this process of grouping clients *client clustering*). In general, the metric estimation component measures various metrics needed by other components of the replica hosting system.

Adaptation triggering addresses the question when to adjust or adapt the system configuration. In other words, we define when and how we can detect that λ has drifted too much from λ^*. Consider a flash crowd causing poor client latency. The system must identify such a situation and react, for example, by increasing the number of replicas to handle the increase in the number of requests. Similarly, congestion in a network where a replica is hosted can result in poor accessibility of that replica. The system must identify such a situation and possibly move that replica to another server. The adaptation-triggering mechanisms do not form an input parameter of the objective function. Instead, they form the heart of the feedback element in Figure 2.1, thus indirectly control λ and maintain the system in an acceptable state.

With **replica placement** we address the question where to place replicas. This issue mainly concerns two problems: selection of locations to install replica servers that can host replicas (*replica server placement*) and selection of replica servers to host replicas of a given object (*replica content placement*). The server placement problem must be addressed during the initial infrastructure installation and during the hosting infrastructure upgrading. The replica content placement algorithms are executed to ensure that content placement results in an acceptable value of λ, given a set of replica servers. Replica placement components use metric estimation services to get the value of metrics required by their placement algorithms. For example, our replica placement algorithm presented in Chapter 4 exploits the scalable latency model produced by means of techniques described in Chapter 3. Both *replica server placement* and *replica content placement* form controllable input parameters of the objective function.

With **consistency enforcement** we consider how to keep the replicas of a given object consistent. Although this thesis assumes the replicated documents to be static, we discuss the general principles of consistency enforcement for the sake of completeness. Maintaining consistency among replicas adds overhead to the system, particularly when the application requires strong consistency (meaning clients are intolerant to stale data) and the number of replicas is large. The problem of consistency enforcement is defined as follows. Given certain application consistency requirements, we must decide which *consistency models*, *consistency policies* and *content distribution mechanisms* can meet these requirements. A consistency model dictates the consistency-related properties of content delivered by the systems to its clients. These models define consistency properties of

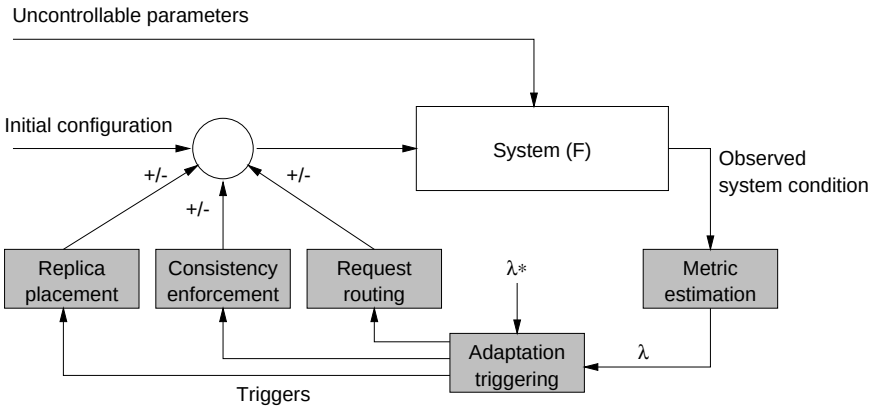


Figure 2.3: Interactions between different components of a wide-area replica hosting system.

objects based on time, value, or the order of transactions executed on the object. A consistency model is usually adopted by consistency policies, which define how, when, and which content distribution mechanisms must be applied. The content distribution mechanisms specify the protocols by which replica servers exchange updates. For example, a system can adopt a time-based consistency model and employ a policy where it guarantees its clients that it will never serve a replica that is more than an hour older than the most recent state of the object. This policy can be enforced by different mechanisms.

Request routing is about deciding how to direct clients to the replicas they need. We choose from a variety of *redirection policies* and *redirection mechanisms*. Whereas the mechanisms provide a method for informing clients about replica locations, the policies are responsible for determining which replica must serve a client. The request routing problem is complementary to the placement problem, as the assumptions made when solving the latter are implemented by the former. For example, we can place replica servers close to our clients, assuming that the redirection policy directs the clients to their nearby replica servers. However, deliberately drifting away from these assumptions can sometimes help in optimizing the objective function. For example, we may decide to direct some client requests to more distant replica servers to offload the client-closest one. Therefore, we treat *request routing* as one of the (controllable) objective function parameters.

Each of the above design issues corresponds to a single *logical* system component. How each of them is actually realized can be very different. The five components together should form a scalable Web replica hosting system. The

interaction between these components is depicted in Figure 2.3, which is a refinement of our initial feedback control system shown in Figure 2.1. We assume that λ^* is a function of the uncontrollable input parameters, that is:

$$\lambda^* = \min_{p_{k+1},\ldots,p_n} F(\underbrace{p_1,\ldots,p_k}_{\substack{\text{Uncontrollable}\\ \text{parameters}}}, \underbrace{p_{k+1},\ldots,p_n}_{\substack{\text{Controllable}\\ \text{parameters}}})$$

Its value is used for adaptation triggering. If the difference with the computed value λ is too high, the triggering component initiates changes in one or more of the three *control components*: replica placement, consistency enforcement, or request routing. These different components strive to maintain λ close to λ^*. They manage the controllable parameters of the objective function, now represented by the actually built system. Of course, the system conditions are also influenced by the uncontrollable parameters. The system condition is measured by the metric estimation services. They produce the current system value λ, which is then passed to the adaptation triggering component for subsequent comparison. This process of adaptation continues throughout the system's lifetime.

Note that the metric estimation services are also being used by components for replica placement, consistency enforcement, and request routing, respectively, for deciding on the quality of their decisions. These interactions are not shown in the figure for sake of clarity.

2.3. METRIC DETERMINATION

All replica hosting systems need to adapt their configuration in an effort to maintain high performance while meeting application constraints at minimum cost. The metric determination component is required to measure the system condition, and thus allow the system to detect when the system quality drifts away from the acceptable interval so that the system can adapt its configuration if necessary.

Another purpose of the metric determination component is to provide each of the three control components with measurements of their input data. For example, replica placement algorithms may need latency measurements in order to generate a placement that is likely to minimize the average latency suffered by the clients. Similarly, consistency enforcement algorithms might require information on object staleness in order to react with switching to stricter consistency mechanisms. Finally, request routing policies may need to know the current load of replica servers in order to distribute the requests currently targeting heavily loaded servers to less loaded ones.

In this section, we discuss three issues that have to be addressed to enable scalable metric determination. The first issue is *metric selection*. Depending on the performance optimization criteria, a number of metrics must be carefully selected to accurately reflect the behavior of the system. Section 2.3.1 discusses metrics related to client latency, network distance, network usage, object hosting cost, and consistency enforcement.

The second issue is *client clustering*. Some client-related metrics should ideally be measured separately for each client. However, this can lead to scalability problems as the number of clients for a typical wide-area replica hosting system can be in the order of millions. A common solution to address this problem is to group clients into clusters and measure client-related metrics on a per-cluster basis. Section 2.3.2 discusses various client clustering schemes.

The third issue is *metric estimation* itself. We must choose mechanisms to collect metric data. These mechanisms typically use client-clustering schemes to estimate client-related metrics. Section 2.3.3 discusses some popular mechanisms that collect metric data.

2.3.1. Choice of metrics

The choice of metrics must allow for evaluating the system performance. First of all, the system must evaluate all metrics that take part in the computation of the objective function. Additionally, the system also needs to measure some extra metrics needed by the control components. For example, a scalable model of network latencies is necessary for running latency-driven replica placement algorithms, although it is not directly used by the cost function.

There exists a wide range of metrics that can reflect the requirements of both the system's clients and the system's operator. For example, the metrics related to latency, distance, and consistency can help evaluate the client-perceived performance. Similarly, the metrics related to network usage and object hosting cost are required to control the overall system maintenance cost, which should remain within bounds defined by the system's operator. We distinguish five classes of metrics, as shown in Figure 2.1, and which are discussed in the following sections.

Temporal metrics

An important class of metrics is related to the *time* it takes for peers to communicate, generally referred to as latency metrics. Latency can be expressed in different ways, depending on what phases of communication are considered. To explain, we consider a client-server system and follow the approach described in

Class	Description
Temporal	The metric reflects how long a certain action takes
Spatial	The metric is expressed in terms of a distance that is related to the topology of the underlying network, or region in which the network lies
Usage	The metric is expressed in terms of usage of resources of the underlying network, notably consumed bandwidth
Financial	Financial metrics are expressed in terms of a monetary unit, reflecting the monetary costs of deploying or using services of the replica hosting system
Consistency	The metrics express to what extent a replica's value may differ from the master copy

Table 2.1: Five different classes of metrics used to evaluate performance in replica hosting systems.

[Dykes et al., 2000] by modeling the total time to process a request, as seen from the client's perspective, as

$$T = T_{DNS} + T_{conn} + T_{res} + T_{rest}$$

T_{DNS} is the DNS lookup time needed to find the server's network address. As reported by Cohen and Kaplan [2001], DNS lookup time can vary tremendously due to cache misses (i.e., the client's local DNS server does not have the address of the requested host in its cache), although in many cases it stays below 500 milliseconds.

T_{conn} is the time needed to establish a TCP connection, which, depending on the type of protocols used in a replica hosting system, may be relevant to take into account. Zari et al. [2001] report that T_{conn} will often be below 200 milliseconds, but that, like in the DNS case, very high values up to even 10 seconds may also occur.

T_{res} is the time needed to transfer a request from the client to the server and receiving the first byte of the response. This metric is comparable to measuring the round-trip time (RTT) between two nodes, but includes the time the server needs to handle the incoming request. Finally, T_{rest} is the time needed to complete the transfer of the entire response.

When considering latency, two different versions are often considered. The *end-to-end latency* is taken as the time needed to send a request to the server, and is often taken as $0.5T_{res}$, possibly including the time T_{conn} to setup a connection. The *client-perceived latency* is defined as $T - T_{rest}$. This latter latency metric reflects the real delay observed by a user.

Obtaining accurate values for latency metrics is not a trivial task as it may require specialized mechanisms, or even a complete infrastructure. One particular problem is predicting client-perceived latency, which not only involves measuring the round-trip delay between two nodes (which is independent of the size of the

response), but may also require measuring bandwidth to determine T_{rest}. The latter has shown to be particularly cumbersome requiring sophisticated techniques [Lai and Baker, 1999]. We return to the problem of latency measurement further in Section 2.3.3.

Spatial metrics

As an alternative to temporal metrics, many systems consider a spatial metric such as number of network-level hops or hops between autonomous systems, or even the geographical distance between two nodes. In these cases, the underlying assumption is generally that there exists a map of the network in which the spatial metric can be expressed.

Maps can have different levels of accuracy. Some maps depict the Internet as a graph of Autonomous Systems (ASes), thus unifying all machines belonging to the same AS. They are used for example by Pierre et al. [2002]. The graph of ASes is relatively simple and easy to operate on. It is also possible to construct highly accurate AS-level graphs by combining the AS-level information extracted from routing tables with that collected using traceroute, reverse DNS lookups, and real-time routing update messages [Mao et al., 2003]. However, because ASes significantly vary in size and internal topology, crossing each of them can affect the characteristics of the transit traffic in any possible way. As a result, even perfectly accurate AS-level graphs cannot provide perfect information about global paths in terms of metrics other than AS-hop count.

Other maps depict the Internet as a graph of routers, thus unifying all machines connected to the same router [Pansiot and Grad, 1998]. These maps are more detailed than the AS-based ones, but are not satisfying predictors for latency. For example, Huffaker et al. [2002] found that the number of router hops is accurate in selecting the closest server in only 60% of the cases. Also, counting router hops can be expected to be increasingly difficult in the future with the wide adoption of such techniques as tunneling or MPLS [Pepelnjak and Guichard, 2001], which may hide the actual routing paths within a network from the hop-counting software. Still, some systems have developed proprietary distance calculation schemes by combining the AS- with router-level network graphs [Rabinovich and Aggarwal, 1999].

Huffaker et al. [2002] examined to what extent geographical distance could be used instead of latency metrics. They showed that there is generally a close correlation between geographical distance and RTT. An earlier study using simple network measurement tools by Ballintijn et al. [2000], however, reported only a weak correlation between geographical distance and RTT. This difference may be caused by the fact that many more monitoring points *outside* the U.S. were used, but that many physical connections actually cross through networks located *in*

the U.S. This phenomenon also caused large deviations in the results presented in [Huffaker et al., 2002].

An interesting approach based on geographical distance is followed in the Global Network Positioning (GNP) project [Ng and Zhang, 2002]. In this case, the Internet is modeled as an N-dimensional geometric space. GNP is used to estimate the latency between two arbitrary nodes. We describe GNP and its several variants in more detail in Chapter 3 when discussing our GNP-based techniques for latency estimation.

Constructing and exploiting a map of the Internet is easier than running an infrastructure for latency measurements. The maps can be derived, for example, from routing tables. Interestingly, Crovella and Carter [1995] reported that the correlation between the distance in terms of hops and the latency is quite poor. However, other studies show that the situation has changed. McManus [1999] shows that the number of hops between ASes can be used as an indicator for client-perceived latency. Research reported in [Obraczka and Silva, 2000] revealed that the correlation between the number of network-level or AS-level hops and round-trip times (RTTs) has further increased, but that RTT still remains the best choice when a single latency metric is needed for measuring client-perceived performance. This is the primary reason why we base our latency estimates on pure latency models, and do not consider other network distance metrics such as the number of hops.

Network usage metrics

Another important metric is the total amount of consumed network resources. The common approach here is to consider only the amount of client traffic. Ideally, however, one should also take into account the utilization of routers and other network elements that participate in traffic handling. As replication brings content closer to the clients, fewer network elements are involved in delivering a single document. The overall network usage expressed in terms of utilized network elements is therefore likely to decrease in comparison to the non-replicated case.

We classify network usage into two types. Internal usage is caused by the communication between replica servers willing to update their replicas. External usage is caused by communication between clients and replica servers. Preferably, the ratio between external and internal usage is high, as internal usage can be viewed as a form of overhead introduced merely to keep replicas consistent.

In order to compare the two types of network usage, consider a non-replicated document of size S bytes that is requested R times per seconds. The total client traffic in this case is $R \cdot S$, and it is transmitted over over some average network distance D_{NR}. When expressed as the number of hops, D_{NR} denotes the number of network elements involved in the transmission of non-replicated content. In

that case, we can express the total traffic processed by all the network elements together as $R \cdot S \cdot D_{NR}$.

On the other hand, suppose the W updates per second are necessary to keep all the replicas consistent. If the average network distance between the server and a replica is D_W, then the update costs are $W \cdot S \cdot D_W$. However, the average network distance when reading a document will now be lower in comparison to the non-replicated case: $R \cdot S \cdot D_R$, where $D_R \ll D_{NR}$. The ultimate impact of replication on network usage depends therefore not only on the average network distances D_{NR} and D_R, but also on the ratio between R and W. For example, if we assume that $D_R = D_W$ and $D_R + D_W = D_{NR}$, then the total network usage changes by the factor of $\frac{W}{R}$ in comparison to the non-replicated case.

Of course, more precise models should be applied in this case, but the example illustrates that merely measuring the amount of client traffic may not be enough to properly determine network usage. This aspect becomes even more important given that replica hosting systems increasingly often need to face issues related to network utilization pricing, an aspect that we discuss next.

Financial metrics

Of a completely different nature are metrics that deal with the economics of content delivery networks. To date, such metrics form a relatively unexplored area, although there is clearly interest to increase our insight (see, for example, [Janiga et al., 2001]). We need to distinguish at least two different roles. First, the owner of the hosting service is confronted with costs for developing and maintaining the hosting service. In particular, costs will be concerned with server placement, server capacity, and network resources (see, e.g., [Chandra et al., 2001]). This calls for metrics aimed at the hosting service provider.

The second role is that of customers of the hosting service. Considering that we are dealing with shared resources that are managed by service provider, accounting management by which a precise record of resource consumption is developed, is important for billing customers [Aboba et al., 2000]. However, developing pricing models is not trivial and it may turn out that simple pricing schemes will dominate the sophisticated ones, even if application of the latter are cheaper for customers [Odlyzko, 2001]. For example, Akamai uses peak consumed bandwidth as its pricing metric.

The pricing model for hosting an object can directly affect the control components. For example, a model can mandate that the number of replicas of an object is constrained by the money paid by the object owner. Likewise, there exist various models that help in determining object hosting costs. Examples include a model with a flat base fee and a price linearly increasing along with the number of object replicas, and a model charging for the total number of clients serviced by

all the object replicas. Finally, yet another study shows how to calculate the cost of hosting an object on other replica hosting systems in an environment where multiple such systems cooperate in order to jointly handle traffic surges [Amini et al., 2004].

Consistency metrics

Consistency metrics inform to what extent the replicas retrieved by the clients are consistent with the replica version that was up-to-date at the moment of retrieval. Many consistency metrics have been proposed and are currently in use. They are usually quantified along three different axes.

In *time-based* consistency models, the difference between two replicas *A* and *B* is measured as the time between the latest update on *A* and the one on *B*. In effect, time-based models measure the staleness of a replica in comparison to another, more recently updated replica. Taking time as a consistency metric is popular in Web-hosting systems as it is easy to implement and independent of the semantics of the replicated object. Because updates are generally performed at only a single primary copy from where they are propagated to secondaries, it is easy to associate a single timestamp with each update and to subsequently measure the staleness of a replica.

In *value-based* models, it is assumed that each replica has an associated numerical value that represents its current content. Consistency is then measured as the numerical difference between two replicas. This metric requires that the semantics of the replicated object are known or otherwise it would be impossible to associate and compare object values. An example of where value-based metrics can be applied is a stock-market Web document containing the current values of shares. In such a case, we could define a Web document to be inconsistent if at least one of the displayed shares differs by more than 2% with the most recent value.

Finally, in *order-based* models, reads and writes are perceived as transactions and replicas can differ only in the order of execution of write transactions according to certain constraints. These constraints can be defined as the allowed number of out-of-order transactions, but can also be based on the dependencies between transactions as is commonly the case for distributed shared-memory systems [Mosberger, 1993], or client-centric consistency models as introduced in Bayou [Terry et al., 1994].

Metric classification

Metrics can be classified into two types: static and dynamic. Static metrics are those whose estimates do not vary with time, as opposed to dynamic metrics. Met-

rics such as the geographical distance are static in nature, whereas metrics such as end-to-end latency, number of router hops or network usage are dynamic. The estimation of dynamic metrics can be a difficult problem as it must be performed regularly to be accurate. Note that determining how often a metric should be estimated is a problem by itself. For example, Paxson [1997a] found that the time periods over which end-to-end routes persist vary from seconds to days, whereas network latencies between hosts can change at any moment.

Dynamic metrics can be more useful when selecting a replica for a given client as they estimate the current situation. For example, Crovella and Carter [1995] conclude that the use of a dynamic metric instead of a static one is more useful for replica selection as the former can also account for dynamic factors such as network congestion. Static metrics, in turn, are likely to be exploited by replica server placement algorithms as they tend to be more directed toward a global, long-lasting situation than an instantaneous one.

In general, however, any combination of metrics can be used by any control component. For example, the placement algorithms proposed by Radoslavov et al. [2001] and Qiu et al. [2001] use dynamic metrics (end-to-end latency and network usage). Also Dilley et al. [2002] and Rabinovich and Aggarwal [1999] use end-to-end latency as a primary metric for determining the replica location. Finally, the request-routing algorithm described in [Szymaniak et al., 2003] exploits network distance measurements. We observe that the existing systems tend to support a small set of metrics, and use all of them in each control component. This is also the approach we follow in this thesis, as most of our solutions either provide or rely on latency information.

2.3.2. Client clustering

As we noticed before, some metrics should be ideally measured on a per-client basis. In a wide-area replica hosting system, for which we can expect millions of clients, this poses a scalability problem to the estimation services as well as the components that need to use them. Hence, there is a need for scalable mechanisms for metric estimation.

A popular approach by which scalability is achieved is *client clustering* in which clients are grouped into clusters. Metrics are then estimated on a per-cluster basis instead of on a per-client basis. Although this solution allows to estimate metrics in a scalable manner, the efficiency of the estimation depends on the accuracy of clustering mechanisms. The underlying assumption here is that the metric value computed for a cluster is representative of values that would be computed for each individual client in that cluster. Accurate clustering schemes are those which keep this assumption valid.

The choice of a clustering scheme depends on the metric it aims to estimate. Below, we present different kinds of clustering schemes that have been proposed in the literature.

Local name servers

Each Internet client contacts its local DNS server to resolve a service host name to its IP address(es). The clustering scheme based on local name servers unifies clients contacting the same name server, as they are assumed to be located in the same network-topological region. This is a useful abstraction as DNS-based request-routing schemes are already used in the Internet. However, the success of these schemes relies on the assumption that clients and local name servers are close to each other. Shaikh et al. [2001] performed a study on the proximity of clients and their name servers based on the HTTP logs from several commercial Web sites. Their study concludes that many clients are at least eight hops away from their representative name servers. The authors also measured the round trip times both from the name servers to the servers (name-server latency) and from the clients to the servers (client latency). It turns out that the correlation between the name-server latency and the actual client latency is quite poor. They conclude that the latency measurements to the name servers are only a weak approximation of the latency to actual clients. These findings have been confirmed by [Mao et al., 2002].

Autonomous Systems

The Internet has been built as a graph of individual network domains, called Autonomous Systems (ASes). The AS clustering scheme groups together clients located in the same AS, as is done, for example, by [Pierre et al., 2002]. This scheme naturally matches the AS-based distance metric. Further clustering can be achieved by grouping ASes into a hierarchy, as proposed by Barford et al. [2001], which in turn can be used to place caches.

Although an AS is usually formed out of a set of networks belonging to a single administrative domain, it does not necessarily mean that these networks are proximal to each other. Therefore, estimating latencies with an AS-based clustering scheme can lead to poor results. Furthermore, since ASes are global in scope, multiple ASes may cover the same geographical area. It is often the case that some IP hosts are very close to each other (either in terms of latency or hops) but belong to different ASes, while other IP hosts are very far apart but belong to the same AS. This makes the AS-based clustering schemes not very effective for proximity-based metric estimations.

Client proxies

In some cases, clients connect to the Internet through *proxies*, which provide them with services such as Web caching and prefetching. Client proxy-based clustering schemes group together all clients using the same proxy into a single cluster. Proxy-based schemes can be useful to measure latency if the clients are close to their proxy servers. An important problem with this scheme is that many clients in the Internet do not use proxies at all. Thus, this clustering scheme will create many clusters consisting of only a single client, which is inefficient with respect to achieving scalability for metric estimation.

Network-aware clustering

Researchers have proposed another scheme for clustering Web clients, which is based on client-network characteristics. Krishnamurthy and Wang [2000] evaluate the effectiveness of a simple mechanism that groups clients having the same first three bytes of their IP addresses into a single cluster. The advantage of this mechanism is that it is fast and very easy to implement. However, this simple mechanism fails in more than 50% of the cases when checking whether grouped clients actually belong to the same network. The authors identify two reasons for failure. First, their scheme wrongly merges small clusters that share the same first three bytes of IP addresses as a single class-C network. Second, it splits several class-A, class-B, and CIDR networks into multiple class-C networks. Therefore, the authors propose a novel method to identify clusters by using the prefixes and network masks information extracted from the Border Gateway Protocol routing tables [Rekhter and Li, 1995]. The proposed mechanism consists of the following steps:

1. Creating a merged prefix table from routing table snapshots
2. Performing the longest prefix matching on each client IP address (as routers do) using the constructed prefix table
3. Classifying all the clients which have the same longest prefix into a single cluster.

The authors demonstrate the effectiveness of their approach by showing a success rate of 99.99% in their validation tests.

Hierarchical clustering

Most clustering schemes aim at achieving a scalable manner of metric estimation. However, if the clusters are too coarse grained, it decreases the accuracy of measurement simply because the underlying assumption that the difference between

the metric estimated to a cluster and to a client is negligible is no longer valid. Hierarchical clustering schemes help in estimating metrics at different levels (such as intra-cluster and inter-cluster), thereby aiming at improving the accuracy of measurement, as in IDMaps [Francis et al., 2001] and Radar [Rabinovich and Aggarwal, 1999]. Performing metric estimations at different levels results not only in better accuracy, but also in better scalability.

Note that there can be other possible schemes of client clustering, based not only on the clients' network addresses or their geographical proximities, but also on their content interests (see, e.g., Xiao and Zhang [2001]). However, such clustering is not primarily related to improving scalability through replication, for which reason we further exclude it from our study.

The ultimate selection of clustering schemes used in our solutions is driven by two factors: efficiency and accuracy. We decided to rely on the simple and efficient class-C clustering in our scalable latency-estimation system described in Chapter 3, as it needs to perform latency measurements at high speeds. However, whenever time constraints are not crucial, we chose to use the most accurate network-aware clustering. For example, we exploit it in our replica placement algorithm presented in Chapter 4. Finally, employing DNS redirection implicitly means that we cluster clients based on their local name servers, just like relying on Web browsers to perform latency measurements forces us to tolerate clustering based on client proxies. However, using these two schemes is a consequence of positioning our solutions in the Web environment rather than a design choice.

2.3.3. Metric estimation schemes

Once the clients are grouped into their respective clusters, the next step is to obtain the values for metrics (such as latency or network overhead). Estimation of metrics on a wide area scale such as the Internet is not a trivial task and has been addressed by several research initiatives before [Francis et al., 2001; Moore and Swany, 1999]. In this section, we discuss the challenges involved in obtaining the value for these metrics.

Metric estimation services are responsible for providing values for the various metrics required by the system. These services aid the control components in taking their decisions. For example, these services can provide replica placement algorithms with a map of the Internet. Also, metric estimation services can use client-clustering schemes to achieve scalability.

Metric estimations schemes can be divided into two groups: *active* and *passive* schemes. Active schemes obtain respective metric data by simulating clients and measuring the performance observed by these simulated clients. Active schemes are usually highly accurate, but these simulations introduce additional load to the replica hosting system. Examples of active mechanisms are Cprobes [Carter and

Crovella, 1997] and Packet Bunch Mode [Paxson, 1997b]. Passive mechanisms obtain the metric data from observations of existing system behavior. Passive schemes do not introduce additional load to the network, but deriving the metric data from past events can suffer from poor accuracy. Examples of passive mechanisms include SPAND [Stemm et al., 2000] and EtE [Fu et al., 2002].

Different metrics are by nature estimated in different manners. For example, metric estimation services are commonly used to measure client latency or network distance. The consistency-related metrics are not measured by a separate metric estimation service, but are usually measured by instrumenting client applications. In this section, our discussion of existing research efforts mainly covers services that estimate network-related metrics.

IDMaps

IDMaps is an active service that aims at providing an architecture for measuring and disseminating distance information across the Internet [Francis et al., 1999, 2001]. IDMaps uses programs called tracers that collect and advertise distance information as so-called distance maps. IDMaps builds its own client-clustering scheme. It groups different geographical regions as boxes and constructs distance maps between these boxes. The number of boxes in the Internet is relatively small (in the order of thousands). Therefore, building a distance table between these boxes is inexpensive. To measure client-server distance, an IDMaps client must calculate the distance to its own box and the distance from the target server to this server's box. Given these two calculations, and the distance between the boxes calculated based on distance maps, the client can discover its real distance to the server. It must be noted that the efficiency of IDMaps heavily depends on the size and placement of boxes.

King

King is an active metric estimation method [Gummadi et al., 2002]. It exploits the global infrastructure of DNS servers to measure the latency between two arbitrary hosts. King approximates the latency between two hosts, H_1 and H_2, with the latency between their local DNS servers, S_1 and S_2.

Assume that a host X needs to calculate the latency between hosts H_1 and H_2. The latency between their local DNS servers, $L_{S_1S_2}$, is calculated based on round-trip times (RTTs) of two DNS queries. With the first query, host X queries the DNS server S_1 about some non-existing DNS name that belongs to a domain hosted by S_1 [see Figure 2.4(a)]. In this way, X discovers its latency to S_1:

$$L_{XS_1} = \frac{1}{2} RTT_1$$

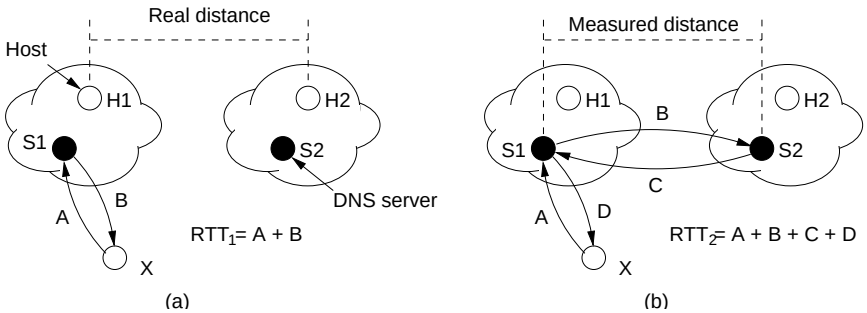


Figure 2.4: The two DNS queries of King

By querying about a non-existing name, X ensures that the response is retrieved from S_1, as no cached copy of that response can be found anywhere in the DNS.

With the second query, host X queries the DNS server S_1 about another non-existing DNS name that this time belongs to a domain hosted by S_2 [see Figure 2.4(b)]. In this way, X measures the latency of its route to S_2 that goes through S_1:

$$L_{XS_2} = \frac{1}{2}RTT_2$$

A crucial observation is that this latency is a sum of two partial latencies, one between X and S_1, and the other between S_1 and S_2: $L_{XS_2} = L_{XS_1} + L_{S_1S_2}$ Since L_{XS_1} has been measured by the first DNS query, X may subtract it from the total latency L_{XS_2} to determine the latency between the DNS servers:

$$L_{S_1S_2} = L_{XS_2} - L_{XS_1} = \frac{1}{2}RTT_2 - \frac{1}{2}RTT_1$$

Note that S_1 will forward the second query to S_2 only if S_1 is configured to accept so-called "recursive" queries from X [Mockapetris, 1987b].

In essence, King is actively probing with DNS queries. A potential problem with this approach is that an extensive use of King may result in overloading the global infrastructure of DNS servers. In that case, the efficiency of DNS is likely to decrease, which can degrade the performance of the entire Internet. Also, according to the DNS specification, it is recommended to reject recursive DNS queries that come from nonlocal clients, which renders many DNS servers unusable for King [Mockapetris, 1987a].

SPAND

SPAND is a shared passive network performance measurement service [Stemm et al., 2000]. This service aims at providing network-related measures such as

client end-to-end latency, available bandwidth, or even application-specific performance details such as access time for a Web object. The components of SPAND are client applications that can log their performance details, a packet-capturing host that logs performance details for SPAND-unaware clients, and performance servers that process the logs sent by the above two components. The performance servers can reply to queries concerning various network-related and application-specific metrics. SPAND has an advantage of being able to produce accurate application-specific metrics if there are several clients using that application in the same shared network. Further, since it employs passive measurement, it does not introduce any additional traffic.

Network Weather Service

The Network Weather Service (NWS) is an active measurement service [Wolski et al., 1999]. It is primarily used in Grid computing, where decisions regarding scheduling of distributed computations are made based on the knowledge of server loads and several network performance metrics, such as available bandwidth and end-to-end latency. Apart from measuring these metrics, it also employs prediction mechanisms to forecast their value based on past events. In NWS, the metrics are measured using special sensor processes, deployed on every potential server node. Further, to measure end-to-end latency active probes are sent between these sensors. NWS uses an adaptive forecasting approach, in which the service dynamically identifies the model that gives the least prediction error. NWS has also been used for replica selection [McCune and Andresen, 1998]. However, exploiting NWS directly by a wide-area replica hosting system can be difficult, as this service does not scale to the level of the Internet. This is due to the fact that it runs sensors in every node and does not use any explicit client clustering schemes. On the other hand, when combined with a good client clustering scheme and careful sensor placement, NWS may become a useful general metric estimation service.

Akamai metric estimation

Commercial replica hosting systems often use their own monitoring or metric estimation services. Akamai has built its own distributed monitoring service to monitor server resources, network-related metrics and overall system performance. The monitoring system places monitors in every replica server to measure server resources. The monitoring system simulates clients to determine if the overall system performance is in an acceptable state as perceived by clients. It measures network-related information by employing agents that communicate with border routers in the Internet as peers and derive the distance-related metrics to be used for its placement decisions [Dilley et al., 2002].

Network positioning

The idea of network positioning has been proposed in [Ng and Zhang, 2002], where it is called Global Network Positioning (GNP). GNP is a novel approach to the problem of network distance estimation, where the Internet is modeled as an N-dimensional geometric space. GNP approximates the latency between any two hosts as the Euclidean distance between their corresponding positions in the geometric space.

GNP relies on the assumption that latencies can be triangulated in the Internet. The position of any host X is computed based on its measured latencies between X and k *landmark* hosts, whose positions have been computed earlier ($k \geq N+1$, to ensure that the calculated position is unique). By treating these latencies as distances, GNP triangulates the position of X. The triangulation is implemented by means of Simplex-downhill, which is a classical optimization method for multi-dimensional functions [Nelder and Mead, 1965]. We discuss the details of the network positioning concept in Chapter 3, when describing our proposed techniques for latency estimation.

Other systems

In addition to the above wide-area metric estimation systems, there are different classes of systems that measure service-related metrics such as content popularity, client-aborted transfers, and amount of consumed bandwidth. These kinds of systems perform estimation in a smaller scale, and mostly measure metrics as seen by a single server.

Web page instrumentation and associated code (e.g., in JavaScript) is being used in various commercial tools for measuring service-related metrics. In these schemes, instrumentation code is downloaded by the client browser after which it tracks the download time for individual objects and reports performance characteristics to the Web site.

EtE is a passive system used for measuring metrics such as access latency, and content popularity for the contents hosted by a server [Fu et al., 2002]. This is done by running a special model near the analyzed server that monitors all the service-related traffic. It is capable of determining sources of delay (distinguishing between network and server delays), content popularity, client-aborted transfers and the impact of remote caching on the server performance.

There also exist a number of user tools for end-to-end bandwidth estimation. Although none of these tools evolved into a full-fledge metric estimation system to date, nothing prevents their exploited techniques from being incorporated by Web hosting systems. A comparative analysis of such user tools can be found in [Shriram et al., 2005], which evaluates their accuracy in a high-speed wide-

area network. The results indicate that those offering the highest accuracy are pathload [Jain and Dovrolis, 2002] and pathchirp [Ribeiro et al., 2003], both of which perform advanced analysis of packet dynamics in the monitored network.

A more general approach has recently been followed in DipZoom [Rabinovich et al., 2006]. The system aims at providing focused, on-demand Internet measurements for large scale distributed systems. Compared to previous solutions that require a well-established infrastructure to perform measurements, DipZoom offers a matchmaking service that allows different measurement providers to trade their services. It uses peer-to-peer techniques to bring together applications in need of measurements with external measurement providers. It then harnesses market forces to orchestrate the supply and demand to provide a "free market" eco-system. This approach can potentially be useful for applications that do not have the luxury of controlling a well-established infrastructure for metric estimation.

2.3.4. Discussion

In this section, we discussed three issues related to metric determination: metric selection, client clustering, and metric estimation.

Metric selection deals with deciding which metrics are important to evaluate system performance. In most cases, optimizing latency is considered to be most important, and we propose a number of techniques for latency estimation in Chapter 3. However, it is also possible to use simpler spatial metrics as long as one assumes that, for example, a low number of network-level hops between two nodes also implies a relatively low latency between those two nodes. Spatial metrics are typically easier to measure, but they tend to be fairly inaccurate estimators of actual latencies.

An alternative metric is consumed bandwidth, which is also used to measure the efficiency of a system. However, in order to measure the efficiency of a consistency protocol as expressed by the ratio between the consumed bandwidth for replica updates and the bandwidth delivered to clients, some distance metric needs to be taken into account as well. When it comes to consistency metrics, three different types need to be considered: those related to time, value, and the ordering of operations. It appears that this differentiation is fairly complete, leaving the actual implementation of consistency models and the enforcement of consistency the main problem to solve.

An interesting observation is that hardly no systems today use financial metrics to evaluate system performance. Designing and applying such metrics is not obvious, but extremely important in the context of system evaluation.

A scalability problem that these metrics introduce is that they, in theory, require measurements on a per-client basis. With millions of potential clients,

such measurements are impossible. Therefore, it is necessary to come to client-clustering schemes. The goal is to design a cluster such that measurement at a single client is representative for any client in that cluster (in a mathematical sense, the clients should form an equivalence class). Finding the right metric for clustering, and also one that can be easily established has shown to be difficult. However, network-aware clustering by which a prefix of the network address is taken as criterion for clustering has lead to very accurate results. This is one of the clustering techniques that we use in our own solutions discussed in this thesis.

Once a metric has been chosen, its value needs to be determined. This is where metric estimation services come into place. Various services exist, including some very recent ones that can handle difficult problems such as estimating the latency between two arbitrary nodes in the Internet. Again, metric estimation services can turn out to be rather complicated, partly due to the lack of sufficient acquisition points in the Internet. In this sense, the approach to model the Internet as a Euclidean N-dimensional space is very powerful as it allows *local computations* concerning remote nodes. However, this approach can be applied only where the modeled metric can be triangulated, making it more difficult when measuring, for example, bandwidth. Still, we demonstrate in this thesis that it is possible to optimize on both latency and bandwidth by combining mechanisms based on network positioning and bandwidth aggregation, as discussed in Chapter 5.

2.4. ADAPTATION TRIGGERING

The performance of a replica hosting system changes with the variations of uncontrollable system parameters such as client access patterns and network conditions. These changes make the current value of the system's λ drift away from the optimal value λ^*, and fall out of the acceptable interval. The system needs to maintain a desired level of performance by keeping λ in an acceptable range amidst these changes. The adaptation triggering component of the system is responsible for identifying changes in the system and for adapting the system configuration to bound λ within the acceptable range. This adaptation consists of a combination of changes in replica placement, consistency policy, and request routing policy.

We classify adaptation triggering components along two criteria. First, they can be classified based on their timing nature. Second, they can be classified based on which element of the system actually performs the triggering.

2.4.1. Time-based classification

Taking timing into account, we distinguish three different triggering mechanisms: periodic triggers, aperiodic triggers, and triggers that combine these two.

Periodic triggers

A periodic triggering component analyzes a number of input variables, or λ itself, at fixed time intervals. If the analysis reveals that λ is too far from λ^*, the system triggers the adaptation. Otherwise, it allows the system to continue with the same configuration. Such a periodic evaluation scheme can be effective for systems that have relatively stable uncontrollable parameters. However, if the uncontrollable parameters fluctuate a lot, then it may become very hard to determine a good evaluation periodicity. A too short period will lead to considerable adaptation overhead, whereas a too long period will result in slow reactions to important changes.

Aperiodic triggers

Aperiodic triggers can trigger adaptation at any time. A trigger is usually due to an event indicating a possible drift of λ from the acceptable interval. Such events are often defined as changes of the uncontrollable parameters, such as the client request rates or end-to-end latency, which may reflect issues that the system has to deal with.

The primary advantage of aperiodic triggers is their responsiveness to emergency situations such as flash crowds where the system must be adapted quickly. However, it requires continuous monitoring of metrics that can indicate events in the system, such as server load or client request rate.

Hybrid triggers

Periodic and aperiodic triggers have opposite qualities and drawbacks. Periodic triggers are well suited for detecting slow changes in the system that aperiodic triggers may not detect, whereas aperiodic triggers are well suited to detect emergency situations where immediate action is required. Consequently, a good approach may be a combination of periodic and aperiodic triggering schemes. For example, Radar and Globule use both periodic and aperiodic triggers, which give them the ability to perform global optimizations and to react to emergency situations.

In Radar [Rabinovich and Aggarwal, 1999], every replica server periodically runs a replication algorithm that checks for the number of client accesses to a particular replica and server load. An object is deleted for low client accesses

and a migration/replication component is invoked if the server load is above a threshold. In addition, a replica server can detect that it is overloaded and ask its replication-managing component to offload it. Adaptation in this case consists either of distributing the load over other replica servers, or to propagate the request to another replication-managing component in case there are not enough replica servers available.

In Globule [Pierre and van Steen, 2006], each primary server periodically evaluates recent client access logs. The need for adapting the replication and consistency policies is determined by this evaluation. Similarly to Radar, each replica server also monitors its request rate and response times. When a server is overloaded, it can request its primary server to reevaluate the replication strategy.

2.4.2. Source-based classification

Adaptation triggering mechanisms also vary upon which part of the system actually performs the triggering. We describe three different kinds of mechanisms.

Server-triggered adaptation

Server-triggered adaptation schemes consider that replica servers are in a good position to evaluate metrics from the system. Therefore, the decision that adaptation is required is taken by one or more replica servers. Radar and Globule use server-triggered adaptation, as they make replica servers monitor and possibly react to system conditions.

Server-triggered adaptation is also well suited for reacting to internal server conditions, such as overloads resulting from flash crowds or denial-of-service (DoS) attacks. For example, Jung et al. [2002] studied the characteristics of flash crowds and DoS attacks. They propose an adaptation scheme where servers can differentiate these two kinds of events and react accordingly: increase the number of replicas in case of a flash crowd, or invoke security mechanisms in case of a DoS attack.

Server-triggered adaptation is effective as the servers are in a good position to determine the need for changes in their strategies in view of other constraints, such as total system cost. Also, these mechanisms do not require running triggering components on elements (hosts, routers) that may not be under the control of the replica hosting system. In our solutions, we assume that the servers retain total control over what happens in the system. In particular, a number of our proposed techniques are transparent to the clients such that they are completely unaware of many operations performed on the server side.

Client-triggered adaptation

Adaptation can be triggered by the clients. In client-triggered schemes, clients or client representatives can notice that they experience poor quality of service and request the system to take the appropriate measures. Sayal et al. [2003] describe such a system where smart clients provide the servers with feedback information to help take replication decisions.

Client-triggered adaptation can be efficient in terms of preserving a client's QoS. However, it has three important drawbacks. First, the clients or client representatives must cooperate with the replica hosting system. Second, client transparency is lost, as clients or their representatives need to monitor events and take explicit action. Third, by relying on individual clients to trigger adaptation, this scheme may suffer from poor scalability in a wide-area network, unless efficient client clustering methods are used.

Router-triggered adaptation

In router-triggered schemes, adaptation is initiated by the routers that can inform the system of network congestion, link and network failures, or degraded end-to-end request latencies. These schemes observe network-related metrics and operate on them.

Such an adaptation scheme is used in SPREAD [Rodriguez and Sibal, 2000]. In SPREAD, every network has one special router with a distinguished proxy attached to it. If the router notices a TCP communication from a client to retrieve data from a primary server, it intercepts this communication and redirects the request to the proxy attached to it. The proxy gets a copy of the referenced object from the primary server and services this client and all future requests passing through its network. By using the network layer to implement replication, this scheme builds an architecture that is transparent to the client.

Router-triggered schemes have the advantage that routers are in a good position to observe network-related metrics, such as end-to-end latency and consumed bandwidth while preserving client transparency. Such schemes are useful to detect network congestion or dead links, and thus may trigger changes in replica location. However, they suffer from two disadvantages. First, they require the support of routers, which may not be available to every enterprise building a replica hosting system in the Internet. Second, they introduce an overhead to the network infrastructure, as they need to isolate the traffic targeting Web hosting systems, which involves processing all packets received by the routers.

2.4.3. Discussion

Deciding when to trigger system adaptation is difficult because explicitly computing λ and λ^* may be expensive, if not impossible. This calls for schemes that are both responsive enough to detect the drift of λ from the acceptable interval and are computationally inexpensive. This is usually realized by monitoring simple metrics which are believed to significantly influence λ.

Another difficulty is posed by the fact that it is not obvious which adaptive components should be triggered. Depending on the origin of the performance drift, the optimal adaptation may be any combination of changes in replica placement, request routing or consistency policies.

2.5. REPLICA PLACEMENT

The task of replica placement algorithms is to find good locations to host replicas. As noted earlier, replica placement forms a controllable input parameter of the objective function. Changes in uncontrollable parameters, such as client request rate or client latencies, may warrant changing the replica locations. In such case, the adaptation triggering component triggers the replica placement algorithms, which subsequently adapt the current placement to new conditions.

The problem of replica placement consists of two subproblems: *replica server placement* and *replica content placement*. Replica server placement is the problem of finding suitable locations for replica servers. Replica content placement is the problem of selecting replica servers that should host replicas of an object. Both these placements can be adjusted by the system to optimize the objective function value λ.

There are some fundamental differences between the server and content placement problems. Server placement concerns the selection of locations that are good for hosting replicas of *many* objects, whereas content placement deals with the selection of locations that are good for replicas of a *single* object. Furthermore, these two problems differ in how often and by whom their respective solutions need to be applied. The server placement algorithms are used in a larger time scale than the content placement algorithms. They are usually used by the system operator during installation of server infrastructure or while upgrading the hosting infrastructure, and can typically be run once every few months. The content placement algorithms are run more often, as they need to react to possibly rapidly changing situations such as flash crowds. This is why it is so important that content placement algorithms have low computational complexity, which allows them to return results within a short time.

Low complexity was one of our goals when designing the latency-driven placement algorithm presented in Chapter 4. The unique property of this algorithm is that it deliberately does not stipulate whether it considers locations to be candidates for replica servers or for the content itself. This enables one to apply our algorithm to both server- and content placement.

We note that Karlsson et al. [2002] present a framework for evaluating replica placement algorithms for content delivery networks and also in other fields such as distributed file systems and databases. Their framework can be used to classify and qualitatively compare the performance of various algorithms using a generic set of primitives covering problem definition and heuristics. They also provide an analytical model to predict the decision times of each algorithm. Their framework is useful for evaluating the *relative performance* of different replica placement algorithms, and as such, complements the material discussed in this section.

2.5.1. Replica server placement

The problem of replica server placement is to select K servers out of N potential sites such that the objective function is optimized for a given network topology, client population, and access patterns. The objective function used by the server placement algorithms operates on some of the metrics defined in Section 2.3. These metrics may include, for example, client latencies for the objects hosted by the system, or the financial cost of server infrastructure.

The problem of determining the number and locations of replica servers, given a network topology, can be modeled as the center placement problem. Two variants used for modeling it are the facility location problem and the minimum K-median problem. Both these problems are NP-hard. They are defined in [Shmoys et al., 1997; Qiu et al., 2001], and we describe them here again for the sake of completeness.

Facility Location Problem Given a set of candidate server locations i in which the replica servers ("facilities") may be installed, running a server in location i incurs a cost of f_i. Each client j must be assigned to one replica server, incurring a cost of $d_j c_{ij}$ where d_j denotes the demand of the node j, and c_{ij} denotes the distance between i and j. The objective is to find the number and location of replica servers which minimizes the overall cost.

Minimum K-Median Problem Given N candidate server locations, we must select K of them (called "centers"), and then assign each client j to its closest center. A client j assigned to a center i incurs a cost of $d_j c_{ij}$. The goal is to select K centers, so that the overall cost is minimal.

The difference between the minimum K-median problem and the facility location problem is that the former associates no cost with opening a center (as with a facility, which has an operating cost of f_i). Further, in the minimum K-median problem, the number of servers is bounded by K.

Some initial work on the problem of replica server placement has been addressed in [da Cunha, 1997]. However, it has otherwise been seldom addressed by the research community and only few solutions have been proposed.

Li et al. [1999] propose a placement algorithm based on the assumption that the underlying network topologies are trees and solve it using dynamic programming techniques. The algorithm is designed for Web proxy placement but is also relevant to server placement. The algorithm works by dividing a tree T into smaller subtrees T_i; the authors show that the best way to place t proxies is by placing t_i proxies for each T_i such that $\sum t_i = t$. The algorithm is shown to be optimal if the underlying network topology is a tree. However, this algorithm has two limitations: first, it cannot be applied to a wide-area network such as the Internet whose topology is not a tree, and (second it has a high computational complexity of $O(N^3K^2)$ where K is the number of proxies and N is the number of candidate locations. We note that the first limitation of this algorithm is due to its assumption about the presence of a single origin server and henceforth to find servers that can host a target Web service. This allows to construct only a tree topology with this origin server as root. However, a typical Web replica hosting system will host documents from multiple origins, falsifying this assumption. This nature of problem formulation is more relevant for content placement, where every document has a single origin Web server.

Qiu et al. [2001] model the replica placement problem as a minimum K-median problem and propose a greedy algorithm. In each iteration, the algorithm selects one server, which offers the least cost, where cost is defined as the average distance between the server and its clients. In the i^{th} iteration, the algorithm evaluates the cost of hosting a replica at the remaining $N-i+1$ potential sites in the presence of already selected $i-1$ servers. The computational cost of the algorithm is $O(N^2K)$. The authors also present a hot-spot algorithm, in which the replicas are placed close to the clients generating most requests. The computational complexity of the hot-spot algorithm is $N^2 + min(NlogN, NK)$. The authors evaluate the performance of these two algorithms and compare each one with the algorithm proposed in [Li et al., 1999]. Their analysis shows that the greedy algorithm performs better than the other two algorithms and its performance is only 1.1 to 1.5 times worse than the optimal solution. The authors note that the placement algorithms need to incorporate the client topology information and access pattern information, such as client end-to-end distances and request rates.

Radoslavov et al. [2001] propose two replica server placement algorithms that do not require the knowledge of client location but decide on replica location based on the network topology alone. The proposed algorithms are *max router fanout* and *max AS/max router fanout*. The first algorithm selects servers closest to the router having maximum fanout in the network. The second algorithm first selects the Autonomous System (AS) with the highest fanout, and then selects a server within that AS that is closest to the router having maximum fanout. The performance studies show that the second algorithm performs only 1.1 to 1.2 times worse than that of the greedy algorithm proposed in [Qiu et al., 2001]. Based on this, the authors argue that the need for knowledge of client locations is not essential. However, it must be noted that these topology-aware algorithms assume that the clients are uniformly spread throughout the network, which may not be true. If clients are not spread uniformly throughout the network, then the algorithm can select replica servers that are close to routers with highest fanout but distant from most of the clients, resulting in poor client-perceived performance.

2.5.2. Replica content placement

The problem of replica content placement consists of two subproblems: *content placement* and *replica creation*. The first problem concerns the selection of a set of replica servers that must hold the replica of a given object. The second problem concerns the selection of a mechanism to inform a replica server about the creation of new replicas.

Content placement

The content placement problem consists of selecting K out of N replica servers to host replicas of an object, such that the objective function is optimized under a given client access pattern and replica update pattern. The content placement algorithms select replica servers in an effort to improve the quality of service provided to the clients and minimize the object hosting cost.

Similarly to the server placement, the content placement problem can be modeled as the facility location placement. However, such solutions can be computationally expensive making it difficult to be applied to this problem, as the content placement algorithms are run far more often their server-related counterparts. Therefore, existing replica hosting systems exploit simpler solutions.

In Radar [Rabinovich and Aggarwal, 1999], every host runs the replica placement algorithm, which defines two client request rate thresholds: R_{rep} for replica replication, and R_{del} for object deletion, where $R_{del} < R_{rep}$. A document is deleted if its client request rate drops below R_{del}. The document is replicated if its client request rate exceeds R_{rep}. For request rates falling between R_{del} and R_{rep},

documents are migrated to a replica server located closer to clients that issue more than a half of requests. The distance is calculated using a Radar-specific metric called preference paths. These preference paths are computed by the servers based on information periodically extracted from routers.

In SPREAD, the replica servers periodically calculate the expected number of requests for an object. Servers decide to create a local replica if the number of requests exceeds a certain threshold [Rodriguez and Sibal, 2000]. These servers remove a replica if the popularity of the object decreases. If required, the total number of replicas of an object can be restricted by the object owner.

Chen et al. [2002] propose a dynamic replica placement algorithm for scalable content delivery. This algorithm uses a *dissemination-tree*-based infrastructure for content delivery and a peer-to-peer location service provided by Tapestry for locating objects [Zhao et al., 2004]. The algorithm works as follows. It first organizes the replica servers holding replicas of the same object into a load-balanced tree. Then, it starts receiving client requests which target the origin server containing some latency constraints. The origin server services the client if the server's capacity constraints and client's latency constraints are met. If any of these conditions fail, it searches for another server in the dissemination tree that satisfies these two constraints and creates a replica at that server. The algorithm aims to achieve better scalability by quickly locating the objects using the peer-to-peer location service. The algorithm is good in terms of preserving client latency and server capacity constraints. On the other hand, it has a considerable overhead caused by checking QoS requirements for every client request. In the worst case a single client request may result in creating a new replica. This can significantly increase the request servicing time.

Kangasharju et al. [2001] model the content placement problem as an optimization problem. The problem is to place K objects in some of N servers, in an effort to minimize the average number of inter-AS hops a request must traverse to be serviced, meeting the storage constraints of each server. The problem is shown to be NP-complete and three heuristics are proposed to address this problem. The first heuristic uses popularity of an object as the only criterion and every server decides upon the objects it needs to host based on the objects' popularity. The second heuristic uses a cost function defined as a product of object popularity and distance of server from origin server. In this heuristic, each server selects the objects to host as the ones with high cost. The intuition behind this heuristic is that each server hosts objects that are highly popular and also that are far away from their origin server, in an effort to minimize the client latency. This heuristic always tries to minimize the distance of a replica from its origin server oblivious of the presence of other replicas. The third heuristic overcomes this limitation and uses a coordinated replication strategy where replica locations are decided in a

global/coordinated fashion for all objects. This heuristic uses a cost function that is a product of total request rate for a server, popularity, and shortest distance of a server to a copy of the object. The central server selects the object and replica pairs that yield the best cost at every iteration and recomputes the shortest distance between servers for each object. Using simulations, the authors show that the global heuristic outperforms the other two heuristics. The drawback is its high computational complexity.

Replica creation mechanisms

Various mechanisms can be used to inform a replica server about the creation of a new replica that it needs to host. The most widely used mechanisms for this purpose are *pull-based caching* and *push replication*.

In pull-based caching, replica servers are not explicitly informed of the creation of a new replica. When a replica server receives a request for a document it does not own, it treats it as a miss and fetches the replica from the master. As a consequence, the creation of a new replica is delayed until the first request for this replica. This scheme is adopted in Akamai [Dilley et al., 2002]. Note that in this case, pull-based caching is used only as a mechanism for replica creation. The decision to place a replica in that server is taken by the system, when redirecting client requests to replica servers.

In push replication, a replica server is informed of a replica creation by explicitly pushing the replica contents to the server. A similar scheme is used in Globule [Pierre and van Steen, 2006] and Radar [Rabinovich and Aggarwal, 1999].

2.5.3. Discussion

The problem of replica server and content placement is not regularly addressed by the research community. A few works have proposed solution for these problems [Qiu et al., 2001; Radoslavov et al., 2001; Chen et al., 2002; Kangasharju et al., 2001]. Still, choosing the best performing content placement algorithm is not trivial as it depends on the access characteristics of the Web content.Pierre and van Steen [2006] showed there is no single best performing replica placement strategy and it must be selected on a per-document basis based on their individual access patterns. Karlsson and Karamanolis [2004] propose a scheme where different placement heuristics are evaluated off-line and the best performing heuristic is selected on a per-document basis. In [Pierre and van Steen, 2006; Sivasubramanian et al., 2003], the authors propose to perform this heuristic selection dynamically where the system adapts to change in access patterns by switching the documents' replication strategies on-the-fly.

Furthermore, the existing server placement algorithms improve client QoS by minimizing client latency or distance [Qiu et al., 2001; Radoslavov et al., 2001]. Even though client QoS is important to make placement decisions, in practice the selection of replica servers is constrained by administrative reasons, such as business relationship with an ISP, and financial cost for installing a replica server. Such a situation introduces a necessary trade-off between financial cost and performance gain, which are not directly comparable entities. This drives the need for server placement solutions that not only take into account the financial cost of a particular server facility but that can also translate the performance gains into potential monetary benefits. To the best of our knowledge little work has been done in this area, which requires building economic models that translate the performance of a replica hosting system into the monetary profit gained. These kinds of economic models are imperative to enable system designers to make better judgments in server placement and provide server placement solutions that can be applied in practice.

We note that an explicit distinction between server and content placement is generally not made. Rather, work has concentrated on finding server locations to host contents of a single content provider. However, separate solutions for server placement and content placement would be more useful in a replica hosting system, as these systems are intended to host different contents with varying client access patterns. We follow this approach in our solution presented in Chapter 4, where we describe our latency-driven replica placement algorithm. One of its vital characteristics is that it identifies locations for replicas according to their access patterns, regardless of whether placing replica servers or the content itself. This makes our algorithm applicable in many different scenarios, depending on how access patterns are calculated and which variant of the replica placement problem (servers or content) needs to be solved.

2.6. CONSISTENCY ENFORCEMENT

This section discusses a number of popular techniques for consistency enforcement. As noted before, all the solutions proposed in this thesis generally assume the replicated data to be static. We therefore do not investigate in detail how to keep replicas consistent in the next chapters. However, given that the goal of this chapter is to provide an overview of Web replication techniques, it discusses some basic definitions and consistency enforcement techniques for the sake of completeness. Note that a number of solutions for consistency enforcement in large-scale Web systems have been proposed in a recent PhD thesis [Sivasubramanian, 2007]. We refer the interested reader to that thesis for a

detailed description of problems and solutions related to preserving replica consistency.

The consistency enforcement problem concerns selecting *consistency models* and implementing them using various *consistency policies*, which in turn can use several *content distribution mechanisms*. A consistency model is a contract between a replica hosting system and its clients that dictates the consistency-related properties of the content delivered by the system. A consistency policy defines how, when, and to which object replicas the various content distribution mechanisms are applied. For each object, a policy adheres to a certain consistency model defined for that object. A single model can be implemented using different policies. A content distribution mechanism is a method by which replica servers exchange replica updates. It defines in what form replica updates are transferred, who initiates the transfer, and when updates take place.

Although consistency models and mechanisms are usually well defined, choosing a valid one for a given object is a nontrivial task. The selection of a consistency model, policies, and mechanisms must ensure that the required level of consistency (defined by various consistency metrics as discussed in Section 2.3) is met, while keeping the communication overhead as low as possible.

2.6.1. Consistency models

Consistency models differ in their strictness of enforcing consistency. By strong consistency, we mean that the system guarantees that all replicas are identical from the perspective of the system's clients. If a given replica is not consistent with others, it cannot be accessed by clients until it is brought up to date. Due to high replica synchronization costs, strong consistency is seldom used in wide-area systems. Weak consistency, in turn, allows replicas to differ, but ensures that all updates reach all replicas after some (bounded) time. Since this model is resistant to delays in update propagation and incurs less synchronization overhead, it fits better in wide-area systems.

Single versus multiple master

Depending on whether updates originate from a single site or from several ones, consistency models can be classified as single-master or multi-master, respectively. The single-master models define one machine to be responsible for holding an up-to-date version of a given object. These models are simple and fit well with applications where the objects by nature have a single source of changes. They are also commonly used in existing replica hosting systems, as these systems usually deliver some centrally managed data. For example, Radar assumes that most of its objects are static Web objects that are modified rarely and uses

primary-copy mechanisms for enforcing consistency. The multi-master models allow more than one server to modify the state of an object. These models are applicable to replicated Web objects whose state can be modified as a result of client access. However, these models introduce new problems such as the necessity of solving update conflicts. Little work has been done on multi-master models in the context of Web replica hosting systems.

Types of consistency

As we explained, consistency models usually define consistency along three different axes: time, value, and order.

Time-based consistency models were formalized in [Torres-Rojas et al., 1999] and define consistency based on real time. These models require a content distribution mechanism to ensure that an update to a replica at time t is visible to the other replicas and clients before time $t + \Delta$. Cate [1992] adopts a time-based consistency model for maintaining consistency of FTP caches. The consistency policy in this system guarantees that the only updates that might not yet be reflected in a site are the ones that have happened in the last 10% of the reported age of the file. Time-based consistency is applicable to all kinds of objects. It can be enforced using different content distribution mechanisms such as *polling* (where a client or replica polls often to see if there is an update), or server invalidation (where a server invalidates a copy held by other replicas and clients if it gets updated). These mechanisms are explained in detail in the next section.

Value-based consistency schemes ensure that the difference between the value of a replica and that of other replicas (and its clients) is no greater than a certain Δ. Value-based schemes can be applied only to objects that have a precise definition of value. For example, an object encompassing the details about the number of seats booked in an aircraft can use such a model. This scheme can be implemented by using polling or server invalidation mechanisms. Examples of value-based consistency schemes and content distribution mechanisms can be found in [Bhide et al., 2002].

Order-based consistency schemes are generally exploited in replicated databases. These models perceive every read/write operation as a transaction and allow the replicas to operate in different states if the out-of-order transactions adhere to the rules defined by these policies. For example, Krishnakumar and Bernstein [1994] introduce the concept of N-ignorant transactions, where a transaction can be carried out on a replica while it is ignorant of N prior transactions in other replicas. The rules constraining the order of execution of transactions can also be defined based on dependencies among transactions. Implementing order-based consistency policies requires content distribution mechanisms to exchange the transactions among all replicas, and transactions need to be timestamped using

mechanisms such as logical clocks [Raynal and Singhal, 1996]. This consistency scheme is applicable to a group of objects that jointly constitute a regularly updated database.

A continuous consistency model, integrating the above three schemes, is presented by Yu and Vahdat [2002]. The underlying premise of this model is that there is a continuum between strong and weak consistency models that is semantically meaningful for a wide range of replicated services, as opposed to traditional consistency models, which explore either strong or weak consistency [Bernstein and Goodman, 1983]. The authors explore the space between these two extremes by making applications specify their desired level of consistency using *conits*. A conit is defined as a unit of consistency. The model uses a three-dimensional vector to quantify consistency: *(numerical error, order error, staleness)*. *Numerical error* is used to define and implement value-based consistency, *order error* is used to define and implement order-based consistency schemes, and *staleness* is used for time-based consistency. If each of these metrics is bound to zero, then the model implements strong consistency. Similarly, if there are no bounds then the model does not provide any consistency at all. The conit-based model allows a broad range of applications to express their consistency requirements. Also, it can precisely describe guarantees or bounds with respect to differences between replicas on a per-replica basis. This enables replicas having poor network connectivity to implement relaxed consistency, whereas replicas with better connectivity can still benefit from stronger consistency. The mechanisms implementing this conit-based model are described in [Yu and Vahdat, 2000] and [Yu and Vahdat, 2002].

2.6.2. Content distribution mechanisms

Content distribution mechanisms define how replica servers exchange updates. These mechanisms differ on two aspects: the forms of the update and the direction in which updates are triggered (from source of update to other replicas or vice versa). The decision about these two aspects influences the system's attainable level of consistency as well as the communication overhead introduced to maintain consistency.

Update forms

Replica updates can be transferred in three different forms. In the first form, called *state shipping*, a whole replica is sent. The advantage of this approach is its simplicity. On the other hand, it may incur significant communication overhead, especially noticeable when a small update is performed on a large object.

In the second update form, called *delta shipping*, only differences with the previous state are transmitted. It generally incurs less communication overhead compared to state shipping, but it requires each replica server to have the previous replica version available. Furthermore, delta shipping assumes that the differences between two object versions can be quickly computed.

In the third update form, called *function shipping*, replica servers exchange the actions that cause the changes. It generally incurs the least communication overhead as the size of description of the action is usually independent from the object state and size. However, it forces each replica server to convey a certain, possibly computationally demanding, operation.

The update form is usually dictated by the exploited replication scheme and the object characteristics. For example, in *active replication* requests targeting an object are processed by all the replicas of this object. In such a case, function shipping is the only choice. In *passive replication*, in turn, requests are first processed by a single replica, and then the remaining ones are brought up-to-date. In such a case, the update form selection depends on the object characteristics and the change itself: whether the object structure allows for changes to be easily expressed as an operation (which suggests function shipping), whether the object size is large compared to the size of the changed part (which suggests delta shipping), and finally, whether the object was simply replaced with a completely new version (which suggests state shipping).

In general, it is the job of a system designer to select the update form that minimizes the overall communication overhead. In most replica hosting systems, updating means simply replacing the whole replica with its new version. However, it has been shown that updating Web objects using delta shipping could reduce the communication overhead by up to 22% compared to commonly used state shipping [Mogul et al., 1997].

Update direction

The update transfer can be initiated either by a replica server that is in need of a new version and wants to *pull* it from one of its peers, or by the replica server that holds a new replica version and wants to *push* it to its peers. It is also possible to combine both mechanisms.

Pull In one version of the pull-based approach, every piece of data is associated with a *Time To Refresh* (TTR) attribute, which denotes the next time the data should be validated. The value of TTR can be a constant, or can be calculated from the update rate of the data. It may also depend on the consistency requirements of the system. Data with high update rates and strong consistency requirements

require a small TTR, whereas data with less updates can have a large TTR. Such a mechanism is used in [Cate, 1992]. The advantage of the pull-based scheme is that it does not require replica servers to store state information, offering the benefit of higher fault tolerance. On the other hand, enforcing stricter consistency depends on careful estimation of TTR: small TTR values will provide good consistency, but at the cost of unnecessary transfers when the document was not updated.

In another pull-based approach, HTTP requests targeting an object are extended with the HTTP *if-modified-since* field. This field contains the modification date of a cached copy of the object. Upon receiving such a request, a Web server compares this date with the modification date of the original object. If the Web server holds a newer version, the entire object is sent as the response. Otherwise, only a header is sent, notifying that the cached copy is still valid. This approach allows for implementing strong consistency. On the other hand, it can impose large communication overhead, as the object home server has to be contacted for each request, even if the cached copy is valid.

In practice, a combination of TTR and checking the validity of a document at the server is used. Only after the TTR value expires, will the server contact the document's origin server to see whether the cached copy is still valid. If it is still valid, a fresh TTR value is assigned to it and a next validation check is postponed until the TTR value expires again.

Push The push-based scheme ensures that communication occurs only when there is an update. The key advantage of this approach is that it can meet strong consistency requirements without introducing the communication overhead known from the "if-modified-since" approach: since the replica server that initiates the update transfer is aware of changes, it can precisely determine which changes to push and when. An important constraint of the push-based scheme is that the object home server needs to keep track of all replica servers to be informed. Although storing this list may seem costly, it has been shown that it can be done in an efficient way [Cao and Liu, 1998]. A more important problem is that the replica holding the state becomes a potential single point of failure, as the failure of this replica affects the consistency of the system until it is fully recovered.

Push-based content distribution schemes can be associated with leases [Gray and Cheriton, 1989]. In such approaches, a replica server registers its interest in a particular object for an associated lease time. The replica server remains registered at the object home server until the lease time expires. During the lease time, the object home server pushes all updates of the object to the replica server. When the lease expires, the replica server can either consider it as potentially stale or register at the object home server again.

Leases can be divided into three groups: age-based, renewal-frequency-based, and load-based ones [Duvvuri et al., 2000]. In the age-based leases, the lease time depends on the last time the object was modified. The underlying assumption is that objects that have not been modified for a long time will remain unmodified for some time to come. In the renewal-frequency-based leases, the object home server gives longer leases to replica servers that ask for replica validation more often. In this way, the object server prefers replica servers used by clients expressing more interest in the object. Finally, with load-based leases the object home server tends to give away shorter lease times when it becomes overloaded. By doing that, the object home server reduces the number of replica servers to which the object updates have to be pushed, which is expected to reduce the size of the state held at the object home server.

Other schemes The pull and push approaches can be combined in different ways. Bhide et al. [2002] propose three different combination schemes of Push and Pull. The first scheme, called *Push-and-Pull (PaP)*, simultaneously employs push and pull to exchange updates and has tunable parameters to control the extent of push and pulls. The second scheme, *Push-or-Pull (PoP)*, allows a server to adaptively choose between a push- or pull-based approach for each connection. This scheme allows a server to characterize which clients (other replica servers or proxies to which updates need to be propagated) should use either of these two approaches. The characterization can be based on system dynamics. By default, clients are forced to use the pull-based approach. PoP is a more effective solution than PaP, as the server can determine the moment of switching between push and pull, depending on its resource availability. The third scheme, called *PoPoPaP*, is an extended version of PoP, that chooses from Push, Pull and PaP. PoPoPaP improves the resilience of the server (compared to PoP), offers graceful degradation, and can maintain strong consistency.

Another way of combining push and pull is to allow the former to trigger the latter. It can be done either explicitly, by means of *invalidations*, or implicitly, with *versioning*. Invalidations are pushed by an object's origin server to a replica server. They inform the replica server or the clients that the replica it holds is outdated. In case the replica server needs the current version, it pulls it from the origin server. Invalidations may reduce the network overhead, compared to pushing regular updates, as the replica servers do not have to hold the current version for all the time and can delay its retrieval until it is really needed. It is particularly useful for often-updated, rarely-accessed objects.

Versioning techniques are exploited in Akamai [Dilley et al., 2002; Leighton and Lewin, 2000]. In this approach, every object is assigned a unique version,

modified after each update. The parent document that contains a reference to the object is rewritten after each update as well, so that it points to the latest version. The consistency problem is thus reduced to maintaining the consistency of the parent document. Each time a client retrieves the document, the object reference is followed and a replica server is queried for that object. If the replica server notices that it does not have a copy of the referenced version, the new version is pulled in from the origin server.

Scalable mechanisms All the aforesaid content distribution mechanisms do not scale for large number of replicas (say, in the order of thousands). In this case, push-based mechanisms suffer from the overhead of storing the state of each replica and updating them (through separate unicast connections). Pull-based mechanisms suffer from the disadvantage of creating a hot spot around the origin server with thousands of replicas requesting the origin server (again through separate connections) for an update periodically. Both mechanisms suffer from excessive network traffic for updating large number of replicas, as the same updates are sent to different replicas using separate connections. This also introduces considerable overhead on the server, in addition to increasing the network overhead. These scalability limitations require the need for building scalable mechanisms.

Scalable content distribution mechanisms proposed in the literature aim to solve scalability problems of conventional push and pull mechanisms by building a content distribution hierarchy of replicas or clustering objects.

The first approach is adopted in [Ninan et al., 2002; Tewari et al., 2002; Fei, 2001]. In this approach, a content distribution tree of replicas is built for each object. The origin server sends its update only to the root of the tree (instead of the entire set of replicas), which in turn forwards the update to the next level of nodes in the tree and so on. The content distribution tree can be built either using network multicasting or application-level multicasting solutions. This approach drastically reduces the overall amount of data shipped by the origin server. Ninan et al. [2002] propose a scalable lease-based consistency mechanism where leases are made with a replica group (with the same consistency requirement), instead of individual replicas. Each lease group has its own content distribution hierarchy to send their replica updates. Similarly Tewari et al. [2002] propose a mechanism that builds a content distribution hierarchy and also uses object clustering to improve the scalability.

Fei [2001] proposes a mechanism that chooses between update propagation (through a multicast tree) or invalidation schemes on a per-object basis, periodically, based on each object's update and access rate. The basic intuition of the mechanism is to choose propagation if an object is accessed more than it is updated (thereby reducing the pull traffic) and invalidation otherwise (as the over-

head for shipping updates is higher than pulling updates of an object only when it is accessed). The mechanism computes the traffic overhead of the two methods for maintaining consistency of an object, given its past update and access rate. It chooses the one that introduces the least overhead as the mechanism to be adopted for that object.

Object clustering is the process of clustering various objects with similar properties (update and/or request patterns) and treating them as a single clustered object. It reduces the connection initiation overhead during the transmission of replica updates, from a per-object level to per-cluster level, as updates for a cluster are sent in a single connection instead of individuals connection for each object (note the amount of updates transferred using both mechanisms are the same). Clustering also reduces the number of objects to be maintained by a server, which can help in reducing the adaptation overhead as a single decision will affect more objects. To our knowledge, object clustering is not used in any well-known replica hosting system.

2.6.3. Mechanisms for dynamically generated objects

With the development of Web forums, e-commerce sites, blogs and many others, an increasing fraction of Web content is not delivered from a static file but generated dynamically each time a request is received. Dynamic content generation allows servers to deliver personalized Web documents to each user, and to take action when specific requests are issued, such as ordering an item from an e-commerce site.

Dynamic Web applications are often organized along a three-tiered architecture, as depicted in Figure 2.5(a). When a request is issued, the Web server invokes application-specific code, which generates the content to be delivered to the client. This application code, in turn, issues queries to a database where the application state is preserved.

From the point of view of a replica hosting system, it can be tempting to host such Web applications using similar techniques as for static content. One can indeed ignore the fact that documents are dynamically generated, and cache the content as it is generated by the application. However, this technique, called fragment caching, offers poor performance as it is often unlikely that the exact same request will be issued again at the same server. Moreover, maintaining the consistency of dynamic document copies is hard because any update in the underlying database can potentially invalidate a copy.

An improved solution consists of duplicating the application code at all replica servers while the database remains centralized. This allows each server to execute the application in reaction to client requests [see Figure 2.5(b)]. Edge server computing, as it is called, allows servers to generate contents tailored to the specifici-

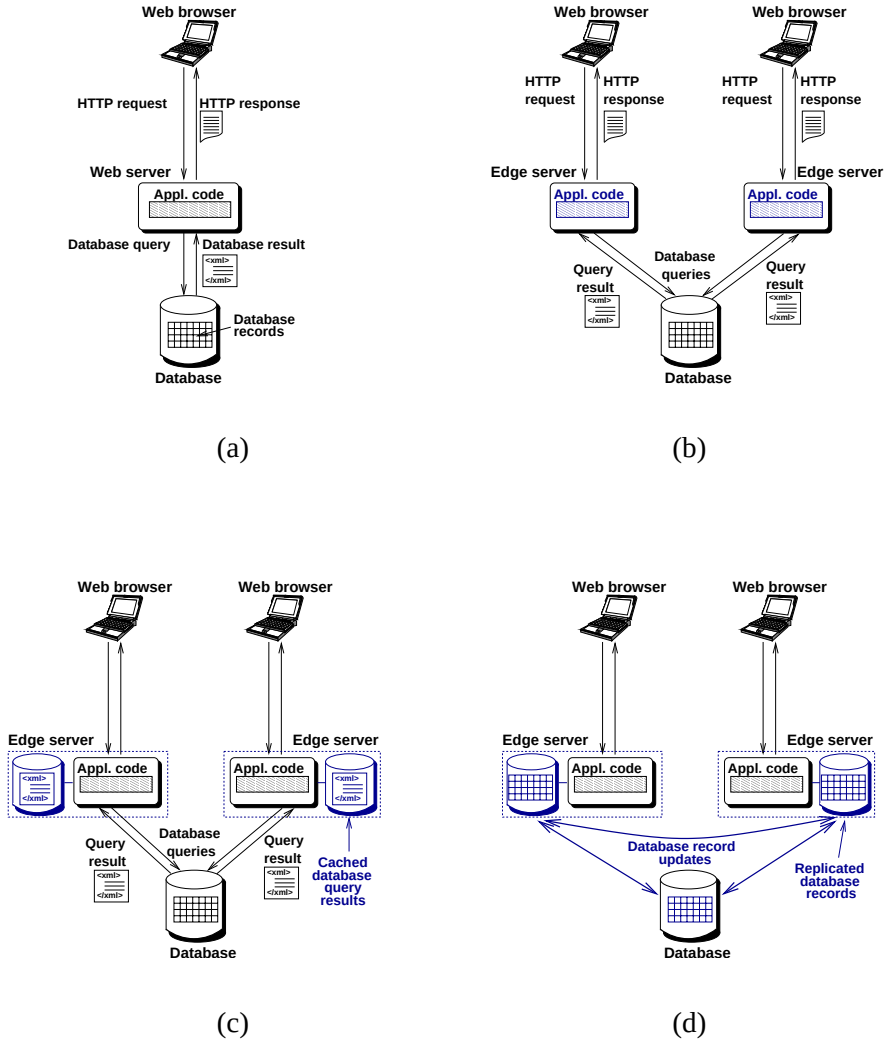


Figure 2.5: Various Web application hosting techniques

ties of each client request while distributing the computational load [Davis et al., 2004]. On the other hand, the centralized database often constitutes a performance bottleneck, which limits the scale that such systems can reach.

To overcome these limitations, it is necessary to move the data to the edge servers, thereby reducing the load of the database. Two types of systems can be distinguished. First, it is possible to cache the results of database queries at the edge servers [see Figure 2.5(c)]. Content-aware caching requires each edge server to run its own database server which contains a partial view of the centralized database [Amiri et al., 2003; Bornhovd et al., 2004]. Each query is subject to a so-called 'query containment check' to determine whether it can be answered from the locally available data. When this is not the case, the query is issued to the central database. Results are subsequently inserted in the local database, before being returned to the application.

A second, simpler alternative to content-aware caching is content-blind caching, where the edge servers do not need to run a database server nor be aware of the database structure [Sivasubramanian et al., 2006]. Instead, it stores query result structures independently from each other. This results in storing potentially redundant information at the edge servers. On the other hand, storing precomputed query results eliminates the database overhead of content-aware caching.

Finally, database caching techniques work well only for applications which repeatedly issue the same queries to their database. For applications which do not exhibit this behavior, it can be more efficient to replicate the whole database at the edge servers [see Figure 2.5(d)]. This guarantees that edge servers can always query their local database copy. On the other hand, database replication involves a lot of communication when the database is updated. One way to deal with this problem is to use partial database replication [Sivasubramanian et al., 2005].

2.6.4. Discussion

In this section, we discussed two important components of consistency enforcement namely, consistency models and content distribution mechanisms. In consistency models, we listed different types of consistency models – based on time, value or transaction orders. In addition to these models, we also discussed the continuous consistency model, in which different network applications can express the consistency constraints in any point in the consistency spectrum. This model is useful to capture the consistency requirements for a broad range of applications being hosted by a replica hosting system. However, the mechanisms proposed to enforce its policies do not scale with increasing number of replicas. Similar models need to be developed for Web replica hosting systems that can provide bounds on inconsistent access of its replicas with no loss of scalability.

In content distribution mechanisms, we discussed the advantages and disadvantages of push, pull, and other adaptive mechanisms. These mechanisms can be broadly classified as server-driven and client-driven consistency mechanisms, depending on who is responsible for enforcing consistency. At the outset, client-driven mechanisms seems to be a more scalable option for large-scale hosting systems, as in this case the server is not overloaded with the responsibility of enforcing consistency. However, in [Yin et al., 2002] the authors have shown that server-driven consistency protocols can meet the scalability requirements of large-scale dynamic Web services delivering both static and dynamic Web content.

Maintaining consistency among dynamic documents requires special mechanisms. Such documents are increasingly often generated by servers organized into a three-tiered architecture that consists of Web servers, application servers, and database servers. However, while such architectures spread the effort of document generation across many machines, they also suffer from the performance bottleneck at the database servers shared by the application servers. To alleviate the impact of this bottleneck, one can cache the results of database queries at the application servers. A detailed discussion of issues arising in that case can be found in [Sivasubramanian, 2007].

2.7. REQUEST ROUTING

In request routing, we address the problem of deciding which replica server shall best service a given client request, in terms of the metrics selected in Section 2.3. These metrics can be, for example, replica server load (where we choose the replica server with the lowest load), end-to-end latency (where we choose the replica server that offers the shortest response time to the client), or distance (where we choose the replica server that is closest to the client).

Selecting a replica is difficult, because the conditions on the replica servers (e.g., load) and in the network (e.g., latency) change continuously. These changing conditions may lead to different replica selections, depending on when and for which client these selections are made. In other words, a replica optimal for a given client may not necessarily remain optimal for the same client forever. Similarly, even if two clients request the same document simultaneously, they may be directed to different replicas. In this section, we refer to these two kinds of conditions as “system conditions.”

As mentioned in Chapter 1, this thesis assumes that client requests are routed in two stages. First, clients are redirected to some *network region* based on network latencies modeled by means of our latency models. Second, they are dynamically pointed to some replica server within that region based on other met-

rics, such as network bandwidth and replica server load. This two-stage strategy enables the system to consider multiple metrics and hence allows it to react to a variety of changes in system conditions.

The entire request routing problem can be split into two: devising a redirection policy and selecting a redirection mechanism. A redirection policy defines how to select a replica in response to a given client request. It is basically an algorithm invoked when the client request is invoked. A redirection mechanism, in turn, is a means of informing the client about this selection. It first invokes a redirection policy, and then provides the client with the redirecting response that the policy generates.

A redirection system can be deployed either on the client side, or on the server side, or somewhere in the network between these two. It is also possible to combine client-side and server-side techniques to achieve better performance [Karaul et al., 1998]. Interestingly, a study by Rodriguez et al. [2000] suggests that clients may easily circumvent the problem of replica selection by simultaneously retrieving their data from several replica servers. This claim is disputed by Kangasharju et al. [2001], who notice that the delay caused by opening connections to multiple servers can outweigh the actual gain in content download time. In this thesis, we assume that we leave the client side unmodified, as the only software that usually works there is a Web browser. We therefore do not discuss the details of client-side server-selection techniques, which can be found in [Conti et al., 2002]. Finally, we do not discuss various Web caching schemes, which have been thoroughly described in [Rodriguez et al., 2001], as caches are by nature deployed on the client-side.

In this section, we examine redirection policies and redirection mechanisms separately. For each of them, we discuss several related research efforts, and summarize with a comparison of these efforts. We also point at the policies and mechanisms that we exploit in our solutions discussed later.

2.7.1. Redirection policies

A redirection policy can be either adaptive or nonadaptive. The former considers current system conditions while selecting a replica, whereas the latter does not. Adaptive redirection policies are usually more complex than nonadaptive ones, but this effort is likely to pay off with higher system performance. The systems we discuss below usually implement both types of policies and can be configured to use any combination of them.

Nonadaptive policies

Nonadaptive redirection policies select a replica that a client should access without monitoring the current system conditions. Instead, they exploit heuristics that assume certain properties of these conditions of which we discuss examples below. Although nonadaptive policies are usually easier to implement, the system works efficiently only when the assumptions made by the heuristics are met.

An example of a nonadaptive policy is round-robin. It aims at balancing the load of replica servers by evenly distributing all the requests among these servers [Delgadillo, 1999; Radware, 2002; Szymaniak et al., 2003]. The assumption here is that all the replica servers have similar processing capabilities, and that any of them can service any client request. This simple policy has proved to work well in clusters, where all the replica servers are located in the same place [Pai et al., 1998]. In wide-area systems, however, replica servers are usually distant from each other. Since round-robin ignores this aspect, it cannot prevent directing client requests to more distant replica servers. If that happens, the client-perceived performance may turn out to be poor. Another problem is that the aim of load balancing itself is not necessarily achieved, as processing different requests can involve significantly different computational costs.

A nonadaptive policy exploited in Radar is the following. All replica servers are ranked according to their predicted load, which is derived from the number of requests each of them has serviced so far. Then, the policy redirects clients so that the load is balanced across the replica servers, and that (additionally) the client-server distance is as low as possible. The assumption here is that the replica server load and the client-server distance are the main factors influencing the efficiency of request processing. Aggarwal and Rabinovich [1998] observe that this simple policy often performs nearly as good as its adaptive counterpart, which we describe below. However, as both of them ignore network congestion, the resulting client-perceived performance may still turn out to be poor.

Several interesting nonadaptive policies were implemented in Cisco DistributedDirector [Delgadillo, 1999]. The first one defines the percentage of all requests that each replica server receives. In this way, it can send more requests to more powerful replica servers and achieve better resource utilization. Another policy allows for defining preferences of one replica server over the other. It may be used to temporarily relieve a replica server from service (for maintenance purposes, for example), and delegate the requests it would normally service to another server. Finally, DistributedDirector enables random request redirection, which can be used for comparisons during some system efficiency tests. Although all these policies are easy to implement, they completely ignore current system conditions, making them inadequate to react to emergency situations.

One can imagine a nonadaptive redirection policy that statically assigns clients to replicas based on their geographical location. The underlying assumptions are that the clients are evenly distributed over the world, and that the geographical distance to a server reflects the network latency to that server. Although the former assumption is not likely to be valid in the general case, the latter has been verified positively as we discussed earlier. According to Huffaker et al. [2002], the correlation between the geographical distance and the network latency reaches up to 75%. Still, since this policy ignores the load of replica servers, it can redirect clients to overloaded replica servers, which may lead to substantially degraded client experience.

Finally, an interesting nonadaptive policy that has later been used in developing the Chord peer-to-peer system, is consistent hashing [Karger et al., 1999]. The idea is straightforward: a URL is hashed to a value h from a large space of identifiers. That value is then used to efficiently route a request along a logical ring consisting of cache servers with IDs from that same space. The cache server with the smallest ID larger than h is responsible for holding copies of the referenced data. Variations of this scheme have been extensively researched in the context of peer-to-peer file sharing systems [Balakrishnan et al., 2003], including those that take network proximity into account (see, e.g., [Castro et al., 2003]).

Adaptive policies

Adaptive redirection policies discover the current system conditions by means of metric estimation mechanisms discussed in Section 2.3. In this way, they are able to adjust their behavior to situations that normally do not occur, like flash crowds, and ensure high system robustness [Wang et al., 2002].

The information that adaptive policies obtain from metric estimation mechanisms may include, for example, the load of replica servers or the congestion of selected network links. Apart from these data, a policy may also need to know some request-related information. The bare minimum is what object is requested and where the client is located. More advanced replica selection can also take client QoS requirements into account.

Knowing the system conditions and the client-related information, adaptive policies first determine a set of replica servers that are capable of handling the request (i.e., they store a replica of the document and can offer required quality of service). Then, these policies select one (or more) of these servers, according to the metrics they exploit. Adaptive policies may exploit more than one metric. More importantly, a selection based on one metric is not necessarily optimal in terms of others. For example, Johnson et al. [2001] observed that most CDNs do not always select the replica server closest to the client.

The adaptive policy used by Globule selects the replica servers that are closest to the client in terms of network distance [Szymaniak et al., 2003]. Globule employs the AS-path length metric, originally proposed by [McManus, 1999], and determines the distance based on a periodically refreshed, AS-based map of the Internet. Since this approach uses passive metric estimation services, it does not introduce any additional traffic to the network. We consider it to be adaptive, because the map of the Internet is periodically rebuilt, which results in (slow) adaptation to network topology changes. Unfortunately, the AS-based distance calculations, although simple to perform, are not very accurate [Huffaker et al., 2002].

A distance-based policy is also implicitly exploited by SPREAD [Rodriguez and Sibal, 2000]. In this system, routers simply intercept requests on their path towards the object home server, and redirect them to a near-by replica server. Consequently, requests reach their closest replica servers, and the resulting client-server paths are shortened. This policy in a natural way adapts to changes in routing. Its biggest disadvantage is the high cost of deployment, as it requires modifying many routers.

A combined policy, considering both replica server load and client-server distance, is implemented in Radar. The policy first isolates the replica servers whose load is below a certain threshold. Then, from these servers, the client-closest one is selected. The Radar redirection policy adapts to changing replica server loads and tries to direct clients to their closest replica servers. However, by ignoring network congestion and end-to-end latencies, Radar focuses more on load balancing than on improving the client-perceived performance.

Adaptive policies that consider client-server latencies have been proposed by Ardaiz et al. [2001] and Andrews et al. [2002]. Based either on the client access logs, or on passive server-side latency measurements, respectively, these policies redirect a client to the replica server that has recently reported the minimal latency to the client. The most important advantage of these schemes is that they exploit latency measurements, which are the best indicator of actual client experience [Huffaker et al., 2002]. On the other hand, both of them require maintaining a central database of measurements, which limits the scalability of systems that exploit these schemes.

A set of adaptive policies is supported by Web Server Director [Radware, 2002]. It monitors the number of clients and the amount of network traffic serviced by each replica server. It also takes advantage of performance metrics specific for Windows NT, which are included in the Management Information Base (MIB). Since this information is only provided in a commercial white paper, it is difficult to evaluate the efficiency of these solutions.

Another set of adaptive policies is implemented in Cisco DistributedDirector [Delgadillo, 1999]. This system supports many different metrics, including inter-AS distance, intra-AS distance, and end-to-end latency. The redirection policy can determine the replica server based on a weighted combination of these three metrics. Although this policy is clearly more flexible than a policy that uses only one metric, measuring all the metrics requires deploying an "agent" on every replica server. Also, the exploited active latency measurements introduce additional traffic to the Internet. Finally, because DistributedDirector is kept separate from the replica servers, it cannot probe their load – it can be approximated only with the nonadaptive policies discussed above.

A complex adaptive policy is used in Akamai [Dilley et al., 2002]. It considers a few additional metrics, like replica server load, the reliability of routes between the client and each of the replica servers, and the bandwidth that is currently available to a replica server. Unfortunately, the actual policy is subject to trade secret and cannot be found in the published literature. However, a recent performance analysis of that policy indicates that the redirecting decisions of Akamai are strongly correlated with network latencies between Web clients and Akamai data centers [Su et al., 2006]. Given that these decisions are publicly available at Akamai's DNS servers, it is possible to exploit the Akamai's redirection policy without knowing its actual algorithm. This can be an attractive option for replicated systems that are unable to develop state-of-the-art redirection policies on their own.

An interesting redirection policy has recently been proposed in Meridian, which is a lightweight, scalable and accurate framework for performing node selection based on network location [Wong et al., 2005]. Meridian organizes nodes into a loosely connected overlay, in which each node monitors a small number of other nodes, especially those in its proximity. All the monitored nodes are organized into concentric rings of exponentially increasing radii. Meridian routes queries to nodes in specific network locations by "zooming-in" onto the target location based on the rings maintained by each node. An important advantage of Meridian is that the targets do not need to belong to the Meridian overlay as long as the network distance between each target and each overlay node can be measured. This allows for deploying Meridian in content delivery networks, in which replica servers can easily be organized into Meridian overlays. In that case, Meridian identifies the closest replica server for each client by zooming-in onto that client while traversing the overlay. The performance results indicate that Meridian performs at least as good as modern redirection policies based on network positioning.

Whereas traditional redirection policies choose individual replica servers for the clients, some systems need to select among *groups* of replica servers [Amini

et al., 2003]. This happens, for example, when multiple content delivery networks (CDNs) cooperate to jointly handle traffic surges. The resulting *peering system* employs redirection policies that choose between complete CDNs rather than between individual replica servers. One such policy relies on a cost-optimized CDN selection algorithm, which translates the CDN selection problem into that of minimum cost network flow optimization [Amini et al., 2004]. Then, it computes a fraction of requests that should be routed to each CDN such that the overall (monetary) cost is minimized without violating the network delay and CDN capacity constraints. The performance measurements based on real-world traces indicate that the proposed CDN selection algorithm performs significantly better than its greedy counterpart in a range of operating environments.

The advantages of adaptive policies are the reason why we prefer them in our solutions over their nonadaptive counterparts. We employ the two-stage concept of redirection, which enables our redirection policy to cover multiple metrics. It first (globally) routes client requests to their respective network regions according to latency, and then (locally) selects individual replica servers within regions based on network bandwidth or server load. Note that our approach clearly prioritizes latency over other metrics, which is in line with the idea of latency-driven replication.

2.7.2. Redirection mechanisms

Redirection mechanisms provide clients with the information generated by the redirection policies. Redirection mechanisms can be classified according to several criteria. For example, Barbir et al. [2002], classify redirection mechanisms into transport-level, DNS-based, and application-level ones. The authors use the term "request routing" to refer to what we call "redirection" in this thesis. Such classification is dictated by the diversity of request processing stages, where redirection can be incorporated: packet routing, name resolution, and application-specific redirection implementation.

In this section, we distinguish between transparent, nontransparent, and combined mechanisms. Transparent redirection mechanisms hide the redirection from the clients. In other words, a client cannot determine which replica server is servicing it. In nontransparent redirection mechanisms, the redirection is visible to the client, which can then explicitly refer to the replica server it is using. Combined redirection mechanisms combine two previous types. They take the best from these two types and eliminate their disadvantages.

As we only focus on wide-area systems, we do not discuss solutions that are applicable only to local environments. An example of such a solution is packet handoff, which is thoroughly discussed in a survey of load-balancing techniques

by Cardellini et al. [1999]. Another survey by the same authors covers other techniques for local-area Web clusters [Cardellini et al., 2002].

Transparent mechanisms

Transparent redirection mechanisms perform client request redirection in a transparent manner. Therefore, they do not introduce explicit bounds between clients and replica servers, even if the clients store references to replicas. It is particularly important for mobile clients and for dynamically changing network environments, as in both these cases, a replica server now optimal for a given client can become suboptimal shortly later.

Several transparent redirection mechanisms are based on DNS [Delgadillo, 1999; Cardellini et al., 2003; Rabinovich and Aggarwal, 1999; Radware, 2002; Szymaniak et al., 2003]. They exploit specially modified DNS servers. When a modified DNS server receives a resolution query for a replicated service, a redirection policy is invoked to generate one or more service IP addresses, which are returned to the client. The policy chooses the replica servers based on the IP address of the query sender. In DNS-based redirection, transparency is achieved assuming that services are referred to by means of their DNS names, and not their IP addresses. The entire redirection mechanism is extremely popular, because of its simplicity and independence from the actual replicated service – as it is incorporated in the name resolution service, it can be used by any Internet application.

On the other hand, DNS-based redirection has some limitations [Shaikh et al., 2001]. The most important ones are poor client identification and coarse redirection granularity. Poor client identification is caused by the fact that a DNS query does not necessarily carry the address of the querying client. The query can pass through several DNS servers before it reaches the one that knows the answer. However, any of these DNS servers knows only the DNS server with which it directly communicates, and not the querying client. Consequently, using DNS-based redirection mechanisms forces the system to use the clustering scheme based on local DNS servers, which was discussed in Section 2.3. The coarse redirection granularity is caused by the granularity of DNS itself: as it deals only with machine names, it can redirect based only on that part of an object URL that is related to the machine name. Therefore, as long as two URLs refer to the same machine name, they are identical for the DNS-based redirection mechanism, which makes it difficult to use different distribution schemes for different objects.

A scalable version of DNS-based redirection is implemented in Akamai. This system improves the scalability of the redirection mechanism by maintaining two groups of DNS servers: top- and low-level ones. Whereas the former share one location, the latter are scattered over several Internet data centers, and are usually

accompanied by replica servers. A top-level DNS servers redirects a client query to a low-level DNS server proximal to the query sender. Then, the low-level DNS server redirects the sender to an actual replica server, usually placed in the same Internet data center. What is important, however, is that the top-to-low level redirection occurs only periodically (about once per hour) and remains valid during all that time. For this reason, the queries are usually handled by proximal low-level DNS servers, which results in short name-resolution latency. Also, because the low-level DNS servers and the replica servers share the same Internet data center, the former may have accurate system condition information about the latter. Therefore, the low-level DNS servers may quickly react to sudden changes, such as flash crowds or replica server failures. These advantages of scalable DNS redirection are the reason why we use it as one of the mechanisms implementing our proposed two-stage redirection, as discussed in Chapter 5.

While DNS mechanisms typically switch clients between replica servers, redirection can also be implemented between different tiers of a multi-tier replicated system. Such systems typically consist of Web servers generating Web pages for the clients, application servers running the actual application logic, and database servers hosting the application data. In that case, Web servers can interface not only with local application servers, but also with those in other datacenters. This idea is explored by WARD, which enables a multi-tier system to run application- and database servers in multiple data centers [Ranjan et al., 2004]. WARD transparently tunnels requests to remote servers when their local counterparts become overloaded. Given that network latencies between data centers are typically very low compared to the time of generating a complex response for a client, the overhead of forwarding requests to remote data centers is negligible. At the same time, spreading the request processing load across multiple data centers reduces the overall response generation time, resulting in a better client-perceived system performance.

An original transparent redirection scheme is exploited in SPREAD, which makes proxies responsible for client redirection [Rodriguez and Sibal, 2000]. SPREAD assumes the existence of a distributed infrastructure of proxies, each handling all HTTP traffic in its neighborhood. Each proxy works as follows. It inspects the HTTP-carrying IP packets and isolates those that are targeting replicated services. All other packets are routed traditionally. If the requested replica is not available locally, the service-related packets are forwarded to another proxy along the path toward the original service site. Otherwise, the proxy services them and generates IP packets carrying the response. The proxy rewrites source addresses in these packets, so that the client thought that the response originates from the original service site. The SPREAD scheme can be perceived as a dis-

tributed packet handoff. It is transparent to the clients, but it requires a whole infrastructure of proxies.

Another novel transparent redirection mechanism is introduced in this thesis. As shall be seen, our mechanism exploits network protocols originally devised for mobile communication, which rely on address-translation capabilities of client machines. We demonstrate that the client-side address translation can be controlled remotely by any Web system supporting the mobile protocols, which effectively enables that system to implement wide-area handoffs. An important advantage of our solution is that it does not introduce any forwarding overhead known from the local handoff schemes (also known as triangular routing), and so it preserves communication efficiency. We present the details of wide-area handoffs in Chapter 5, and demonstrate how our two-stage redirection strategy employs the wide-area handoff to perform replica server selection within network regions in Chapter 5.

Nontransparent mechanisms

Nontransparent redirection mechanisms reveal the redirection to the clients. In this way, these mechanisms introduce an explicit binding between a client and a given replica server. On the other hand, nontransparent redirection mechanisms are easier to implement than their transparent counterparts. They also offer fine redirection granularity (per object), thus allowing for more flexible content management.

The simplest method that gives the effect of nontransparent redirection is to allow a client to choose from a list of available replica servers. This approach is called "manual redirection" and can often be found on Web services of widely-known corporations. However, since this method is entirely manual, it is of little use for replica hosting systems, which require an automated client redirection scheme.

Nontransparent redirection can be implemented with HTTP. It is another redirection mechanism supported by Web Server Director [Radware, 2002]. An HTTP-based mechanism can redirect clients by rewriting object URLs inside HTML documents, so that these URLs point at object replicas stored on some replica servers. It is possible to treat each object URL separately, which allows for using virtually any replica placement. The two biggest advantages of the HTTP-based redirection are flexibility and simplicity. Its biggest drawback is the lack of transparency.

Cisco DistributedDirector also supports the HTTP-based redirection, although in a different manner [Delgadillo, 1999]. Instead of rewriting URLs, this system exploits the HTTP 302 (temporary moved) response code. In this way, the redirecting machine does not need to store any service-related content – all it does is activate the redirection policy and redirect client to a replica server. On the other

hand, this solution can efficiently redirect only per entire Web service, and not per object.

Combined mechanisms

It is possible to combine transparent and nontransparent redirection mechanisms to achieve a better result. Such approaches are followed by Akamai [Dilley et al., 2002], DistributedDirector [Delgadillo, 1999] and Web Server Director [Radware, 2002]. These systems allow to redirect clients using a "cascade" of different redirection mechanisms.

The first mechanism in the cascade is HTTP. A replica server may rewrite URLs inside an HTML document so that the URLs of different embedded objects contain different DNS names. Each DNS name identifies a group of replica servers that store a given object.

Although it is in general not recommended to scatter objects embedded in a single Web page over too many servers [Kangasharju et al., 2001], it may be sometimes beneficial to host objects of different *types* on separate groups of replica servers. For example, as video hosting may require specialized replica server resources, it may be reasonable to serve video streams with dedicated video servers, while providing images with other, regular ones. In such cases, video-related URLs contain a different DNS name (like "video.cdn.com") than the image-related URLs (like "images.cdn.com").

URL rewriting weakens the transparency, as the clients are able to discover that the content is retrieved from different replica servers. However, because the rewritten URLs contain DNS names that point to *groups* of replica servers, the clients are not bound to any *single* replica server. In this way, the system preserves the most important property of transparent redirection systems.

The second mechanism in the cascade is DNS. The DNS redirection system chooses the best replica server within each group by resolving the group-corresponding DNS name. In this way, the same DNS-based mechanism can be used to redirect a client to its several best replica servers, each belonging to a separate group.

By using DNS, the redirection system remains scalable, as it happens in the case of pure DNS-based mechanisms. By combining DNS with URL rewriting, however, the system may offer finer redirection granularity and thus allow for more flexible replica placement strategies.

The third mechanism in the cascade is packet handoff. The processing capabilities of a replica server may be improved by deploying the replica server as a cluster of machines that share the same IP address. In this case, the packet-handoff is implemented locally to scatter client requests across several machines.

Table 2.2: The comparison of representative implementations of a redirection system

System	Redirection																
	Policies											Mechanisms					
	Adaptive						Nonadaptive					TCP		DNS		HTTP	Comb.
	DST	LAT	NLD	CPU	USR	OTH	RR	%RQ	PRF	RND	PLD	CNT	DST	1LV	2LV		
Akamai	X	X	X	X		X									X	X	X
Globule	X						X		X					X			
Radar	X			X							X			X			
SPREAD	X												X				
Cisco DD	X	X					X	X	X	X		X		X		X	X
Web Direct			X		X	X	X					X		X		X	X

DST: Network distance
LAT : End-to-end latency
NLD: Network load
CPU: Replica server CPU load
USR: Number of users
OTH : Other metrics
RR : Round robin
%RQ: Percentage of requests
PRF : Server preference
RND : Random selection
PLD: Predicted load
CNT: Centralized
DST: Distributed
1LV : One-level
2LV : Two-level

Similarly to pure packet-handoff techniques, this part of the redirection cascade remains transparent for the clients. However, since packet handoff is implemented only locally, the scalability of the redirection system is maintained. Note that local packet handoff can also be replaced by our wide-area handoff scheme, which effectively leads to classifying our two-stage redirection mechanism as a combined one.

As can be observed, combining different redirection mechanisms leads to constructing a redirection system that is simultaneously fine-grained, transparent, and scalable. The only potential problem is that deploying and maintaining such a mechanism is a complex task. In practice, however, this problem turns out to be just one more task of a replica hosting system operator. The duties like maintaining a set of reliable replica servers, managing multiple replicas of many objects, and making these replicas consistent, are likely to be at similar (if not higher) level of complexity.

2.7.3. Discussion

The problem of request routing can be divided into two subproblems: devising a redirection policy and selecting a redirection mechanism. The policy decides to which replica a given client should be redirected, whereas the mechanism takes care of delivering this decision to the client.

We classify redirection policies into two groups: adaptive and nonadaptive ones. Nonadaptive policies perform well only when the system conditions do not change. If they do change, the system performance may turn out to be poor. Adaptive policies solve this problem by monitoring the system conditions and adjusting their behavior accordingly. However, they make the system more complex, as they need specialized metric estimation services. We note that all the investigated systems implement both adaptive and nonadaptive policies (see Table 2.2).

We classify redirection mechanisms into three groups: transparent, nontransparent, and combined ones. Transparent mechanisms can be based on DNS or packet handoff. As can be observed in Table 2.2, DNS-based mechanisms are very popular. Among them, a particularly interesting one is the scalable DNS-based mechanism built by Akamai. As for packet handoff, its traditional limitation to clusters can be alleviated by means of a global infrastructure, as is done in SPREAD. Nontransparent mechanisms are based on HTTP. They achieve finer redirection granularity, at the cost of introducing an explicit binding between a client and a replica server. Transparent and nontransparent mechanisms can be combined. Resulting hybrids offer fine-grained, transparent, and scalable redirection at the cost of higher complexity.

We observe that the request routing component has to cooperate with the metric estimation services to work efficiently. Consequently, the quality of the request routing component depends on the accuracy of the data provided by the metric estimation service.

Further, we note that simple, unadaptive policies sometimes work nearly as efficient as their adaptive counterparts. Although this phenomenon may justify using only nonadaptive policies in simple systems, we do believe that monitoring system conditions is of a key value for efficient request routing in large infrastructures. Moreover, combining several different metrics in the process of replica selection may additionally improve the system performance.

Finally, we are convinced that using combined redirection mechanisms is inevitable for large-scale wide-area systems. These mechanisms offer fine-grained, transparent, and scalable redirection at the cost of higher complexity. The resulting complexity, however, is not significantly larger compared to that of other parts of a replica hosting system. Since the ability to support millions of clients can be of fundamental importance, using a combined redirection mechanism is definitely worth the effort. We therefore follow this approach in our two-stage redirection, as discussed in Chapter 5.

2.8. CONCLUSION

In this chapter, we have discussed the most important aspects of replica hosting system development. We have provided a generalized framework for such systems, which consists of five components: metric estimation, adaptation triggering, replica placement, consistency enforcement, and request routing. The framework has been built around an objective function, which allows to formally express the goals of replica hosting system development. For each of the framework components, we have discussed its corresponding problems, described several solutions for them, and reviewed some representative research efforts. Finally, we also discussed how selected elements of hosting system design relate to the techniques proposed in subsequent chapters of this thesis.

The subsequent chapters propose how to improve the response times of replica hosting systems. We believe that these times depend mostly on network latencies between clients and replica servers, as these latencies determine the key characteristics of communication between a replica hosting system and its clients, including the replica access delay and the maximum bandwidth available to a network connection established between a client and a replica server. We split the process of improving client-replica latencies into three parts, each focusing on a different design component of Web replica hosting systems: metric estimation, replica placement, and request routing. The next chapter discusses how to efficiently produce latency models in large-scale distributed systems.

CHAPTER 3

Latency Estimation

3.1. INTRODUCTION

Modern large-scale distributed applications can benefit from information about latencies observed between their various components. Knowing such latencies, a distributed application can organize its operation such that the communication delays between its components are minimized [Zari et al., 2001; Dabek et al., 2004; Pereira et al., 2004]. For example, a content delivery network can place its hosted data such that its clients are serviced at their proximal datacenters [Szymaniak et al., 2006a; Amini and Schulzrinne, 2004]. In addition to improving the client-experienced latency, reducing the overall length of client-to-replica network paths allows one to localize the communication, leading to lower backbone and inter-ISP link utilization. Analogous benefits can be achieved for other large-scale distributed applications such as peer-to-peer overlays or online gaming platforms.

The effectiveness of latency-driven techniques in improving the application performance depends on the accuracy of the latency information. A simple solution consists of periodically probing each latency that the application needs to know [Wolski et al., 1999]. However, such an approach makes sense only in relatively small systems, as continuous probing of pair-wise latencies is clearly not feasible when the number of nodes is very large. For example, redirecting clients to their nearest datacenters would require Google to maintain latency information from virtually every Web client in the Internet to each of its datacenters [Brin and Page, 1998]. Also, the high dynamics of the Internet causes recently measured latencies to not always be a good indication of their current counterparts, as one latency measurement result is not a good predictor of a subsequent identical measurement. These two problems drive the need for scalable and accurate techniques for latency discovery.

A promising approach to the problem of scalable latency estimation is GNP, which models Internet latencies in a multidimensional geometric space [Ng and Zhang, 2002]. Given a small number of "base" latency measurements to a number of dedicated "landmark" nodes, GNP associates each node with its coordinates in that space. The latency between any pair of nodes can then be approximated with the Euclidean distance between their corresponding coordinates. What makes GNP scalable is the constant low number of measurements necessary to position each machine, which enables GNP to estimate all-pair latencies between a large number of machines at low cost.

The attractiveness of GNP has resulted in its various aspects being investigated for several years. However, whereas numerous theoretical properties of GNP have been described in detail [Shavitt and Tankel, 2004; Srinivasan and Zegura, 2004; Tang and Crovella, 2003; Cox et al., 2004; Lim et al., 2003; Pietzuch et al., 2005; Lee et al., 2006], no large-scale GNP implementations have been deployed in real-world environments.

The common property of existing GNP implementations that hinders their deployment is active participation of positioned nodes, which are responsible for measuring and propagating their own base latencies [Castro et al., 2004; Pias et al., 2003; Dabek et al., 2004; Ng and Zhang, 2004]. Such an approach has several disadvantages. First, it introduces problems with malicious nodes lying about their base latencies. Handling such nodes is usually very difficult, and typically comes at the expense of increased system complexity. Second, independent measurements of base latencies performed by many active nodes might overload both the network and the landmarks. This, in turn, might lead to numerous measurement inaccuracies affecting the GNP performance. Finally, active participation requires running some special software at each positioned node. Meanwhile, deploying such software might be infeasible. For example, in content delivery networks, most nodes are unmodifiable third-party Web browsers.

This chapter presents a GNP implementation that addresses all these issues. Our solution is based on two key observations. First, instead of relying on remote nodes to measure and report their base latencies, one can measure these latencies passively on the landmark side. This eliminates the need for customizing the remote nodes and ensures the integrity of measurement results. Second, instead of allowing remote nodes to independently perform their measurements, one can trigger measurements individually using a central, yet scalable, scheduler. This prevents landmarks from overloads and reduces the overall network overhead in general, as the scheduler triggers only the measurements that are really necessary. We demonstrate the feasibility of our approach by incorporating GNP into the content delivery network operated by Google, which enables us to position millions of Google clients.

Compared to the previous GNP implementations, our approach has several advantages. First, it greatly facilitates system deployment, as only the landmarks and the scheduler need to be instrumented. Second, it removes the problem of malicious nodes, as all the instrumented nodes are kept under full control of Google. Third, it eliminates the risk of overloading the landmarks, as the scheduler effectively adjusts the measurement volume to the landmark capacity.

Implementing our system at the scale of millions of clients requires one to address a number of subtle issues. For example, it is necessary to transparently and efficiently schedule measurements such that they do not affect the client-perceived browsing performance. Also, implementing a centralized scheduler is far from trivial when millions of Web clients are serviced by thousands of globally distributed Web servers [Barroso et al., 2003]. Finally, producing GNP coordinates that can remain representative for a long time requires that some special preprocessing techniques are applied to base latencies.

Within the first 2 months of operation, our positioning system performed more than 75 million latency measurements to more than 22 million unique Google clients. Using host clustering techniques allowed us to compute the coordinates of more than 200 million Internet hosts falling into more than 880,000 of /24 networks. To our best knowledge, this is the largest experiment involving network positioning performed so far.

Our study confirms many earlier results, and adds to them by extensively investigating the issue of coordinate stability over time. We show that coordinates drift away from their initial values with time, making 25% of the coordinates to be off by more than 33 milliseconds after one week. However, daily recomputations make 75% of the coordinates stay within 6 milliseconds of their initial values. We also recommend to derive daily coordinates from base latencies measured until around 10pm UTC, as it results in coordinates remaining representative throughout the most of the next 24 hours.

We also contribute to understanding the practical applicability of GNP coordinates in real-life systems. We demonstrate that using latency estimates to decide on client-to-replica redirection leads to selecting replicas closest in term of *measured* latency in 86% of all cases. In another 10% of all cases, clients are redirected to replicas offering latencies that are at most two times longer than optimal. Also, we show that positioning Google clients makes it possible to estimate latencies between globally distributed Internet hosts that have not participate in our measurements. We treat this result as an incentive to develop a new publicly available Google service providing pairwise latency estimates for Internet hosts.

Finally, we address the problem of latency estimation in federated environments such as peer-to-peer networks. We notice that following the centralized approach in such systems is infeasible, as their decentralized nature prevents em-

ploying a centralized scheduler. We therefore propose a fully decentralized scheme in which each host within a federated environment runs its private instance of a positioning system and estimates latencies independently from other hosts. Apart from preserving the decentralized system nature, private spaces enable each node to configure its positioning system instance according to its own requirements. In particular, each host can decide on the selection of landmarks and the particular algorithm used for coordinate computation. We demonstrate that latency estimates performed in different private spaces are highly correlated with each other. This property enables each federated host to run algorithms relying on latency information independently from other hosts, and eliminates the need for a common repository of latencies.

The remainder of this chapter is structured as follows. We discuss a number of related research efforts in Section 3.2. Then follows the description of our system: Section 3.3 describes how we integrated GNP into the Google infrastructure, Section 3.4 shows how to compute stable coordinates, and Section 3.5 discusses our experience with GNP-based client redirection. Section 3.6 evaluates the performance of our system as an application-independent latency estimation service, and Section 3.7 demonstrates how to implement GNP in federated environments in a completely decentralized fashion. Finally, Section 3.8 concludes.

3.2. BACKGROUND

The problem of estimating latencies between all node pairs in the Internet has been investigated for some time. It can be perceived as a special case of a more general problem called traffic matrix estimation, in which one attempts to determine pairwise estimates of any network-related metric such as byte count and packet loss. All such estimates together form a traffic matrix, in which each cell holds a single metric estimate obtained for a given pair of nodes denoted by the matrix coordinates. Traffic matrices are extremely useful in a variety of traffic engineering tasks, including load balancing, network configuration, and capacity planning [Medina et al., 2002].

As the number of estimates carried by a traffic matrix is often very large, they are usually derived from partial information obtained for only a subset of pairs. Several techniques have been proposed to this end, including entropy penalization [Zhang et al., 2003], bayesian inference [Tebaldi and West, 1998], and expectation maximization [Cao et al., 2000]. It has also been shown that traffic matrices can be estimated more accurately when the results of end-to-end pairwise measurements are combined with network-specific information, such as total capacity or network size [Medina et al., 2002]. Interestingly, end-to-end measurements

by themselves already allow for estimating various network properties, including its logical topology [Coates et al., 2002] and per-link packet loss rates [Duffield et al., 2001]. Deriving physical network properties from end-to-end measurements has been commonly referred to as network tomography, which has been shown to perform well in many practical scenarios [Duffield, 2003].

However, there are several important differences between network tomography and network positioning. First, network tomography aims at identifying internal network properties, including per-link characteristics, whereas network positioning solely attempts to estimate end-to-end latencies. It is not concerned with the underlying network topology although small distance between node coordinates has a good chance to indicate their topological proximity. This property makes network positioning particularly suitable for large-scale systems, for which topological information is typically very hard to determine.

Another difference, in a sense related to the first one, is that end-to-end measurements in network tomography are performed systematically in order to decompose the resulting values into parts corresponding to subsequent links in network paths. Such repetitive measurements are likely to incur higher overhead than relatively rare measurements necessary for network positioning. Furthermore, systematic measurements employed by network tomography usually require the cooperation of both communicating parties, whereas network positioning can be implemented transparently to one of these parties, as we demonstrate in this chapter. The transparency of measurements being performed is vital for world-wide deployability of our system, as massive updates of software in the Internet are simply impossible. We believe that all these advantages of network positioning over network tomography make the former more suitable in our case.

3.2.1. Internet node positioning

GNP was the first system to propose modeling the Internet as an N-dimensional geometric space [Ng and Zhang, 2002]. Given such a space, GNP approximates the latency between any pair of hosts as the Euclidean distance between their corresponding coordinates in that space.

The space is determined by the coordinates of "landmark" hosts that GNP computes first. The number of landmarks k must be at least $N+1$ to unambiguously determine the N-dimensional geometric space. Given the landmark coordinates, GNP computes the coordinates of any host X based on the measured latencies between X and each of the landmarks. By treating these latencies as distances, GNP triangulates the coordinates of X relative to the landmark coordinates.

The landmark coordinates are computed as follows. First, GNP instructs the landmarks to measure their latencies to each other. Based on these latencies, GNP calculates all the landmark coordinates so that the distance between any pair of

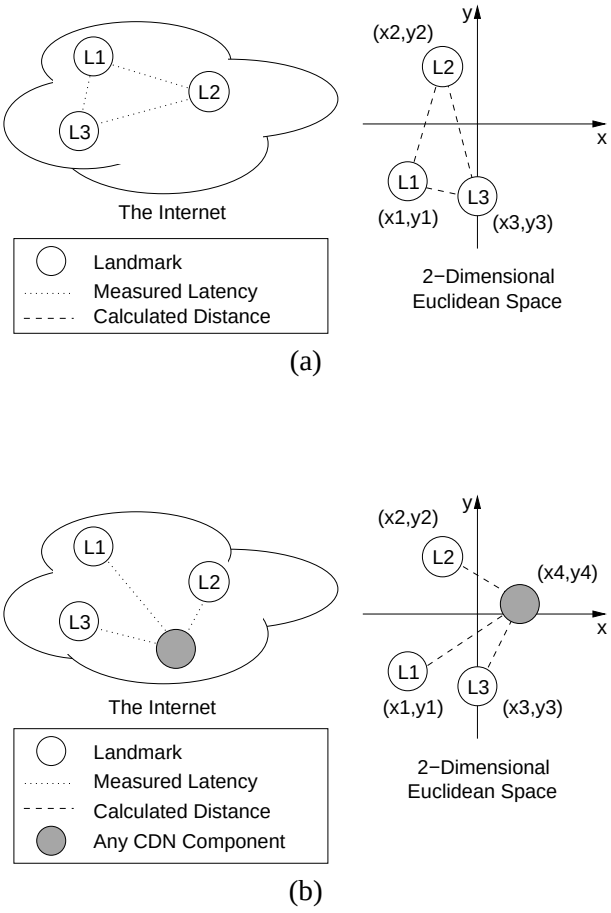


Figure 3.1: GNP: landmark positioning (a), and host positioning (b)

these coordinates is as close as possible to the latency measured between the corresponding pair of the landmarks [see Figure 3.1(a)]. The discrepancy between the distances and their corresponding latencies is minimized using a popular error minimization algorithm called Simplex-downhill [Nelder and Mead, 1965].

Once the landmark coordinates are known, GNP can determine the coordinates of any host X based on the measured latencies between that host and each of the landmarks. The coordinates of X are calculated so that the distance between these coordinates and the coordinates of each landmark is as close as possible to its corresponding measured latency [see Figure 3.1(b)]. This is again achieved by means of the Simplex-downhill algorithm. The GNP authors show that, in 90% of cases, the latency estimations produced by their system are within a relative error ratio of 0.53 compared to the real latency.

3.2.2. Positioning variants

A number of variants have been proposed to the original GNP concept. The PIC project suggested that at least some of the landmarks should be located close to the positioned hosts to improve the positioning accuracy [Castro et al., 2004]. When positioning a global community of Web clients, this suggestion is equivalent to that from another study, which recommends to globally distribute the landmarks in order to achieve higher positioning accuracy [Srinivasan and Zegura, 2004]. We discuss some practical implications of these suggestions in Section 3.3.1.

Another project established that the accuracy and stability of coordinates can be improved by statistical filtering of latency samples used for positioning [Pietzuch et al., 2005]. The intuition is that long-term coordinates should not be affected by temporary and intermittent network conditions such as network congestion. This can be prevented by computing coordinates based on latencies typical for given landmark-host pairs. We verify these findings in our experiments presented in Section 3.4.2. Compared to [Pietzuch et al., 2005], we rely on a much larger and more diverse trace of latencies. We also investigate the issue of how to determine typical latencies, and how often the resulting coordinates need to be recomputed.

The issue of positioning scalability has been addressed in the Lighthouses project [Pias et al., 2003]. It demonstrated that hosts can also be positioned relative to any previously positioned hosts, which in that case act as "local" landmarks. This eliminates the need for measuring latencies to the original landmarks each time a host is positioned, in turn leading to a distribution of the measurement effort resulting in higher positioning scalability. However, as we show below, one can position a huge community of Web clients by relying on the original landmarks only, as long as the measurements performed by the landmarks are appropriately scheduled. This also enables us to avoid the loss of accuracy that using local landmarks inherently incurs.

Other research efforts replace the Simplex-downhill computation used in GNP with simpler optimization schemes [Tang and Crovella, 2003; Lim et al., 2003], or propose alternative distance definitions instead of the Euclidean one [van Langen et al., 2004]. In fact, the selection of a particular positioning algorithm is orthogonal to the question of how to measure latencies required for positioning, as long as all the algorithms require the same set of latencies to be measured. We chose to compute all the coordinates using the Simplex-downhill algorithm recommended in the original GNP paper, as it has performed well when used in our other research projects.

Some of the remaining efforts take a completely different approach and position all hosts simultaneously as a result of a global optimization process [Cox et al., 2004; Shavitt and Tankel, 2004; Dabek et al., 2004; Waldvogel and Rinaldi, 2003]. In that case, there is no need to choose landmarks, since every host is in fact

considered to be a landmark. The respective authors claim that it leads to better accuracy. However, systems relying on the global optimization process generally assume that each host can actively compute its own coordinates based on its latencies measured to other hosts. This assumption makes such an approach applicable only in systems where the operation of each host can be controlled, such as peer-to-peer networks. On the other hand, the global optimization approach is difficult to follow when positioning hosts in a centralized manner. This is because Google cannot generally rely on its clients to measure latencies to each other.

Another approach to latency estimation in peer-to-peer networks (and other federated environments) has been developed in parallel to those relying on a global optimization process [Szymaniak et al., 2004]. It assumes that federated hosts can be organized into independent groups, each running a separate GNP instance with its own, "private" space. As a result, there is no need to negotiate the parameters of a single global space used by all the federated hosts, which relaxes inter-host dependencies and results in better system scalability. We describe and evaluate this approach in greater detail in Section 3.7, when addressing the problem of federated latency estimation.

3.2.3. Positioning implementations

A recent study by the authors of the original GNP paper describes how to implement a global Network Positioning System (NPS) based on GNP [Ng and Zhang, 2004]. The authors identify four key system-building issues that must by addressed by any GNP implementation: maintaining a single global space, adapting to changes in Internet routes, handling fluctuations in network latencies, and computing coordinates as accurately as possible.

NPS addresses the key building issues by organizing hosts interested in positioning into a distributed infrastructure in which each host periodically recalculates its own coordinates. All the coordinates are calculated in the same geometric space, determined by a fixed set of global landmarks. NPS prevents these landmarks from becoming performance bottlenecks by allowing the hosts to position themselves relative not only to the landmarks, but also to any other "reference" hosts whose coordinates are already known. In that sense, NPS generalizes the concept of local landmarks introduced by Lighthouses. On top of that, NPS enables each of the landmarks to compute its coordinates locally by means of a special scheme for decentralized landmark positioning, and exploits some other distributed algorithms to synchronize coordinates computed by different hosts.

The distributed nature of NPS results in improved scalability. However, it also forces NPS to deal with a number of problems that result from the distribution itself, such as preventing malicious hosts from being used as positioning references, synchronizing distributed latency probing to prevent reference hosts

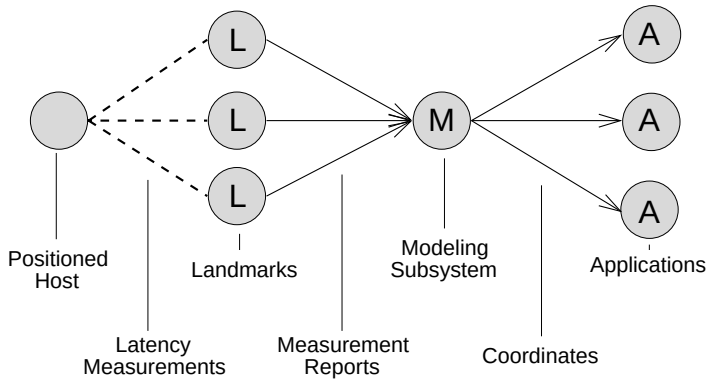


Figure 3.2: The high-level concept of positioning implementation

from being overloaded, or triggering host repositioning to maintain global consistency of coordinates. Solving these problems makes NPS relatively complex. On the other hand, following our centralized approach enables one to avoid all these problems without limiting the system scalability. As a result, our solutions to the four key building issues identified by NPS are much simpler.

3.3. SYSTEM ARCHITECTURE

Using GNP to position Google clients seems to be relatively simple. Essentially, the positioning process can be split into three phases [see Figure 3.2]: measuring base latencies, collecting the measurement results, and modeling latencies in the form of GNP coordinates. The coordinates can then be passed to any latency-driven application, such as those responsible for client redirection or replica placement.

However, as it turns out, naive implementations of either phase in a large-scale Internet service will easily show poor results. This is caused by a number of subtle problems that arise when deploying GNP in a real-world setting. The following sections discuss how we addressed these problems when implementing each phase of the positioning process. We first look into the issue of landmark deployment, including where landmarks should be located and how they should report their measured latencies for modeling. Then, we focus on the mechanisms and policies to implement measurements themselves. Given the huge number of Google clients to be positioned, such measurements must be performed with care to avoid overloading the network. We achieve that by combining distributed triggering of measurements (for scalability) with their centralized scheduling (for tight control). Both triggering and scheduling integrate naturally into the dis-

tributed Google infrastructure without harming its scalability. In particular, the centralized scheduling is performed such that it scales along with the number of measurements that need to be performed. We summarize all our design choices in Section 3.3.4, which presents the final system architecture.

3.3.1. Landmark infrastructure

Landmark deployment

GNP computes the coordinates of each host based on a number of so-called base latencies to that host. Base latencies are measured by landmarks, which must be deployed by the service. Deploying landmarks essentially consists of three steps: deciding on the number of landmarks, on their approximate location, and, finally, on the actual hosting facility where they should be installed.

The first step is to decide on the number of landmarks to deploy. Although GNP is able to compute coordinates using any number of landmarks, previous studies have recommended running at least seven landmarks to obtain good positioning accuracy [Tang and Crovella, 2003; Szymaniak et al., 2004]. Although we use that number of landmarks in our experiments, in practice we also run a number of redundant landmarks to increase the system's resilience to landmark failures.

The second step is to choose approximate geographical locations for the landmarks. As mentioned in Section 3.2, the landmarks should be globally distributed. This is because GNP relies on the assumption that vectors of landmark-to-host latencies are different for hosts located in different parts of the Internet. Should we fail to meet this assumption, then the performance of GNP might turn out to be poor.

To confirm that global landmark distribution is indeed necessary in practice, we evaluated the accuracy of GNP offered by various combinations of landmarks located in different parts of the Internet. To this end, we chose 20 PlanetLab nodes [Bavier et al., 2004] to act as candidate landmarks, and connected them to our positioning system. This allowed us to collect a large set of latencies between the candidate landmarks and a small fraction of Google clients.

The clients in the set turned out to originate from 113 countries, with the number of clients per country varying from 1 to many thousands. To make the evaluation fair for all the countries, we randomly picked 10 clients from each country. For countries represented by less than 10 clients in our trace, all the clients were included. The resulting test set consisted of 616 clients.

Having generated the test set of clients, we iteratively positioned them relative to various combinations of 7 landmarks. We chose this number based on our earlier experiments, where we have shown that 6-dimensional spaces (defined by

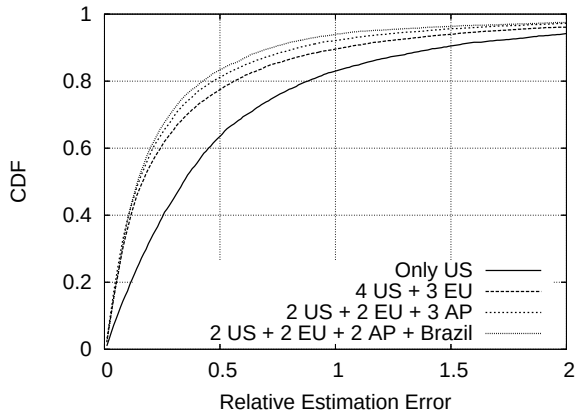


Figure 3.3: The importance of landmark distribution

7 landmarks) are most practical, as estimation accuracy only slightly increases for higher dimensions. The subsequent combinations consisted of manually selected landmarks that were increasingly distributed in a geographical sense. For each combination, we evaluated its offered estimation accuracy based on the latencies measured between the clients and the 13 PlanetLab nodes that were not used for positioning. To this end, we calculated the relative estimation error ε for each such latency similar to GNP:

$$\varepsilon(d_{CL}, d^*_{CL}) = \left| \frac{d^*_{CL} - d_{CL}}{min(d_{CL}, d^*_{CL})} \right|$$

where d_{CL} and d^*_{CL} respectively denote the measured and estimated latencies between client C and landmark L. The distribution of estimation errors observed for four example landmark combinations is depicted in Figure 3.3.

As can be observed, estimation accuracy is lowest when all the landmarks are located in the US. The combination consisting of four American- and three European landmarks offers better accuracy, which improves even further when three of the seven landmarks are located in Asia (Tokyo, Singapore, and China). The best accuracy is offered by the fourth combination, wherein the landmark in Tokyo is replaced with a Brazilian one. This confirms the importance of global landmark distribution, and allows for reaching estimation accuracy close to those reported in the original GNP paper.

The last step of landmark deployment is to choose the actual hosting facilities where the landmarks should be installed. It may seem attractive to deploy landmarks in existing service datacenters to benefit from hardware that is already in place. However, the number or locations of such datacenters may not meet the global landmark distribution requirement. In that case, we need to decouple

the placement of landmarks from the locations of the datacenters by constructing an infrastructure of dedicated landmarks rented from third-party hosting facilities worldwide. In our experiments presented in this chapter, we used the fourth combination of PlanetLab nodes as our landmark set, as it offered the best estimation accuracy.

Latency collector

All the latencies measured by the landmarks must be collected and passed to some modeling component for processing. However, the modeling component typically runs in one of the datacenters. Given that datacenters are normally tightly firewalled, the landmarks deployed outside the datacenters cannot contact the modeling component directly.

One solution to that problem would be to reconfigure the datacenter firewalls to allow incoming traffic from the landmarks. However, doing so potentially exposes the service to attacks initiated from the landmarks. The potential problem becomes even worse when the landmarks are operated by external organizations such as PlanetLab. This solution should therefore be avoided unless there are no other options available.

We therefore decided to follow another approach, in which latencies are collected using network connections opened *from* some dedicated component residing in one of the datacenters *to* the landmarks. This component, called a *collector*, retrieves latencies from the landmarks and stores them in measurement logs accessed by the modeling component. The collector-to-landmark connections are protected with SSL for secure communication.

3.3.2. Latency measurements

Measurement types

Once the landmark infrastructure has been deployed, we can start collecting latencies. There are essentially three kinds of latencies to be measured. First, the landmarks must measure latencies between each other, as GNP requires this information to construct its geometric space. This can easily be achieved by means of periodical active probing, which is the simplest way of discovering latencies between any two machines under our control.

Second, the landmarks must measure their latencies to each datacenter so that the datacenters can be positioned as well. Computing the coordinates of datacenters is necessary to estimate client-datacenter latencies, which can then be used during client redirection. Given that datacenters are operated by the service, the

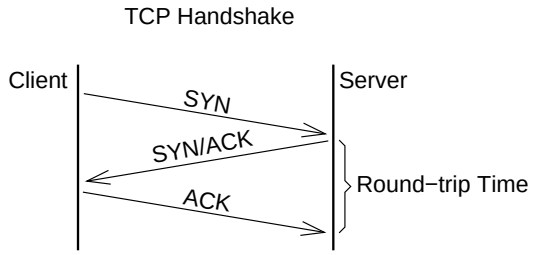


Figure 3.4: Passive latency discovery with SYNACK/ACK

landmarks can discover their latencies to the datacenters by actively probing them just like they probe each other.

Third, the landmarks must determine their latencies to Google clients so that the coordinates of these clients can be computed as well. However, we cannot use active probing this time, as it is likely to trigger various intrusion-detection systems deployed on the client side. This could result in numerous client complaints affecting the service reputation.

Rather than actively probing clients, the landmarks can measure their latencies to the clients without initiating any traffic to these clients. To this end, the landmarks must rely on passive latency discovery, wherein latency measurements can be obtained by monitoring the service traffic and deriving the client latencies from the dynamics of packets constituting that traffic.

A well-known technique for passive latency discovery is the SYNACK/ACK method [Andrews et al., 2002]. It enables a server to estimate its round-trip time to a client when the client initiates a TCP connection to the server. The round-trip time can then be estimated during the TCP handshake phase as the delay between sending the SYNACK packet and receiving its corresponding ACK packet [see Figure 3.4]. We chose this technique for its natural applicability in Web systems, wherein network traffic is typically carried over TCP connections. In principle, however, it is possible to use any other passive latency discovery technique here, such as any of those proposed in [Veal et al., 2005].

Measurement triggering

Using SYNACK/ACK to measure the latency between a client and a landmark requires that the client opens a TCP connection to the landmark. However, the clients issue requests only to datacenters, which are separated from the landmark infrastructure. We must therefore implement a mechanism causing clients to open additional TCP connections to the landmarks.

In general, Google clients are regular Web browsers. A natural way to make them open TCP connections to the landmarks consists of deploying Web servers on the landmarks and instructing the clients to fetch content from these servers.

We can easily instruct Web servers to fetch content from the landmarks by embedding some small landmark-delivered objects inside Google Web pages. A classical example of such objects is a tiny image, which is commonly used by the providers of Web site statistics to track site accesses [SiteMeter.com, 2006]. However, the major drawback with such an approach is that it makes the client experience dependent on the landmark performance, as Web pages can be displayed in their final shape only after all their parts have been retrieved. Datacenters are typically tuned to offer reliable service of high quality to a huge number of clients, but the landmarks are likely to be incomparably less reliable and powerful. Should any landmark face reliability or performance problems, then these failures may become visible to users, and in turn compromise the overall service performance.

Solving this problem requires that the landmark-delivered objects are embedded in such a way that the client-perceived service performance does not depend on the landmarks. In particular, a Web browser should be able to display complete service responses even if the embedded objects cannot be downloaded.

This transparency can be achieved in two ways. First, the service might rely on JavaScript code included in a response to retrieve a number of objects from the landmarks after the response has been displayed [Kokku et al., 2003]. This approach is appealing because JavaScript is supported by most Web browsers. However, the semantics of retrieval failures varies across different JavaScript implementations, which makes it difficult to guarantee that running JavaScript code never results in unexpected browser behavior [Wootton, 1999]. Since one of our priorities was to keep the user's perception of Google untouched, we decided not to risk compromising it by using JavaScript.

Another transparent way of embedding objects is to use server-directed prefetching capabilities of certain browsers [Fisher and Saksena, 2003]. This technique enables a Web server to instruct browsers to retrieve a given object *after* the entire response has been displayed. Prefetching is typically used to accelerate the download of a sequence of Web documents [Duchamp, 1999]. However, it can also be used to trigger the retrieval of landmark-delivered objects.

The service can pass prefetching instructions to Web browsers in the form of special HTTP headers or HTML tags embedded inside its responses [Fielding et al., 1999]. Each such instruction contains the URL of an object that a Web browser should retrieve. In contrast to regular object retrieval, however, Web browsers keep their users unaware of any delays or failures that might occur during prefetching. This guarantees that prefetching does not affect client-perceived service performance.

We decided to employ prefetching to trigger the retrieval of landmark-delivered objects. To this end, we modified Google Web servers to embed prefetching instructions inside their responses such that each tag points at an object hosted by some landmark. This causes the clients to open HTTP connections to the landmarks, which can then perform passive latency discovery.

A potential limitation of prefetching is that it is currently supported only by the Mozilla Firefox Web browser [Firefox, 2006]. This means that Google can only trigger prefetching requests from approximately 11% of its clients [OneStat.com, 2005]. However, prefetching features are planned to be supported by the future releases of Internet Explorer browser as well [Kudallur, 2005]. Also, measuring latencies to a fraction of all the clients might turn out to be enough to position all Internet hosts, as we discuss next.

3.3.3. Measurement scheduling

The above sections have discussed two mechanisms that enable the service to trigger latency measurements: active probing and embedding of prefetching instructions. The configuration of active probing is relatively straightforward, as it involves only a relatively small number of probing targets: landmarks and datacenters. However, deciding on how to trigger a large number of measurements with prefetching is much more difficult.

Obviously, the service needs to trigger all the measurements necessary to position its clients. However, while doing so, it should respect the following three conditions. First, it should trigger only as many measurements as each of the landmarks can handle, as overloaded landmarks cannot measure latencies accurately. Second, it should also keep the total number of measurements low to reduce client-side overhead. Third, it should avoid triggering redundant measurements to minimize network usage.

The following sections describe how our system meets each of these three requirements using a centralized scheduling policy. We then propose how such a policy can be implemented in a large-scale system in which responses are simultaneously generated by the thousands of Web servers that constitute the Google infrastructure [Barroso et al., 2003].

Landmark load

In a naive approach, the service could include prefetching tags in all its responses to perform as many measurements as possible. However, doing so would most likely lead to overloading the network connections to the landmarks, resulting in latencies being measured with high inaccuracies.

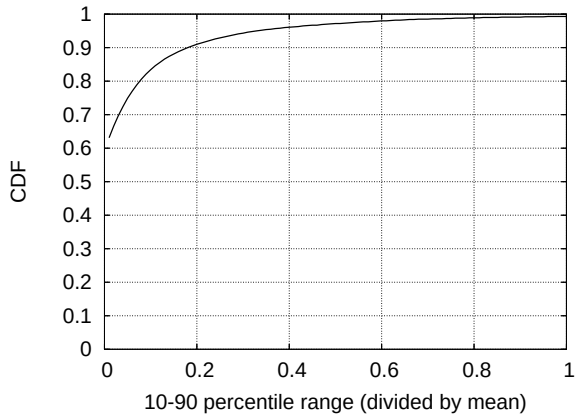


Figure 3.5: Variation of latencies to hosts within a /24 network

Overloading the landmarks can be avoided by limiting the number of measurements performed by each landmark. To this end, the service can enforce some delay between subsequent measurements scheduled to each landmark such that the landmark capacity is never exceeded. The distinguishing property of this time-sharing scheme is that it can be easily distributed over multiple scheduling components, which we benefit from below. It is also very easy to implement, as it only needs to maintain a timestamp of the most recent measurement scheduled to each landmark.

Client clustering

Scheduling individual measurements should ultimately result in collecting all the latencies necessary to position all the clients. However, since the clients might consider measurements to be an unnecessary burden, the service should strive to minimize that burden by reducing the number of measurements.

We decided to reduce the number of measurements issued to the clients by means of clustering, which is a popular technique for reducing the number of operations performed in a distributed system. In principle, clustering groups machines into so-called clusters, and performs the operations on a per-cluster rather than on a per-machine basis. In our case, clustering reduces the number of measurements by grouping clients whose latencies to a given landmark are very similar.

Efficient scheduling requires that clustering is fast, which limits the selection of clustering schemes to very simple ones. An example of such a scheme is clustering of machines whose IP addresses share the same 24-bit prefix. We call each such cluster a /24 network, and identify each such network with its 24-bit prefix.

Given that each /24 network can contain up to 254 machines, /24 clustering can reduce the number of measurements by up to two orders of magnitude.

However, relying on /24 clustering when performing latency measurement is possible only if latencies measured to the clustered machines are similar. To validate whether this condition is met in the Internet, we calculated 10-90 percentile ranges for latencies measured to different clients in the same /24 networks.

The percentile ranges were calculated based on the latency trace collected by our system. First, we extracted latencies measured by the landmark running at MIT during a two-week period. The duration of two weeks was chosen to limit the impact of routing changes on the observed latencies. Second, we identified all the /24 networks containing at least three different clients in the two-week trace. The number of such networks turned out to be 28,540. Third, we obtained an indication of the landmark's latency to each client by calculating a median for each landmark-client pair. Finally, for all the clients in each network, we evaluated how close their median latencies are to each other. To this end, we calculated the 10-90 percentile range over the set of medians, and divided that range by the mean median latency for that network. The resulting distribution of 10-90 percentile range coefficients is depicted in Figure 3.5.

As can be observed, in over 91% of /24 networks, the coefficient of the 10-90 percentile range is lower than 0.2. This means that, in 91% of /24 networks, median latencies to 80% of clients differ by at most 20%. Such a low variation enables the landmarks to measure their latencies to any client in a network, and treat these latencies as representative for any other clients in that network. Note that /24 clustering enables to position all the clients in a given /24 network only if at least one of them supports prefetching. According to our data, this condition is met by about 85% of /24 networks containing Google clients. The remaining clients can be positioned when a more aggressive clustering scheme is used, as we discuss in Section 3.6.

A potential problem related to clustering is posed by client-side routers performing network address translation (NATs) employed by many institutions. They allow for a single IP address being shared by a group of hosts [Huston, 2004]. This is achieved by translating network traffic such that the public IP address visible on the Internet is replaced with private IP addresses valid in the network hidden behind a NAT router, and vice versa. Such translation effectively clusters all the nodes within the hidden network into a single node holding the public IP address.

The problem with NAT is that the size and topology of the hidden network (and even the fact that there is one) remains unknown to the Internet service performing latency measurements. If the topology of the hidden network is complex enough, then the measurements results might vary significantly, potentially causing the median latency to be unrepresentative for many nodes in that network. Fur-

thermore, as the IP addresses of NAT routers already represent clusters of nodes, they intuitively should not be clustered further into /24 networks, especially when many IP addresses within the same /24 network actually belong to NAT routers. However, as NAT routers are by nature hard to detect from the public part of the Internet, there is little one can do about this problem. In our approach, we assume that networks hidden behind NAT routers are either very small or very fast, and do not cause significant variations in latency measurements.

Redundant measurements

Positioning a /24 network requires measuring latencies between that network and all the landmarks. This can be achieved by triggering measurements from a given /24 network to the landmarks in a round-robin fashion. To this end, subsequent service responses sent to each network contain prefetching tags pointing at objects hosted by subsequent landmarks.

A potential problem is that starting all the round-robin sequences from the same landmark is likely to cause that landmark to be fully loaded. In that case, the mechanism responsible for limiting the landmark load will prevent many measurements from being performed. The service can avoid this problem by using random initial landmarks in round-robin sequences specific to different /24 networks.

Another problem with round-robin scheduling is that it keeps triggering measurements from a given network even after a complete set of landmark latencies to that network has been collected. The redundant measurements are of little use to the positioning system and might prevent the service from triggering more useful latencies when the landmark load increases.

We chose to avoid triggering redundant measurements by simply limiting the number of round-robin sessions to a given network. For example, once a complete set of latencies has been collected for a given network, no other measurements are triggered to that network for some time. The duration of the interval between sessions generally depends on how often new coordinates are being computed. In the current setup, we allow only one round-robin session per /24 network every hour.

Scheduling policy

The complete scheduling policy consists of three steps taking place every time a measurement can be triggered to some client. First, the policy determines the client's /24 network by dropping the last 8 bits of the client's IP address.

Next, the policy inspects the round-robin state specific to that network and checks whether any more measurements should be performed to it in its current

round-robin session. If not, then no measurement is triggered. Otherwise, the policy identifies the next landmark that should perform the measurement.

Finally, the policy verifies the approximate load of the selected landmark. If that landmark is currently overloaded, then no measurement is triggered. Otherwise, the policy updates both the round-robin state and the landmark load information, and instructs the service to trigger a measurement between the client and the landmark.

Scheduler separation

Although the scheduling policy is conceptually simple, it is not obvious how to implement it in a distributed Web system such as Google without harming its scalability. This is because it requires the service to maintain a centralized state for both round-robin landmark selection and landmark load approximation. The service needs this information to decide which of its generated Web pages should contain prefetching instructions.

Unfortunately, given the large number and wide-area distribution of Web servers in a large-scale Web system, it is unlikely that they can efficiently share state among them [Barroso et al., 2003]. This is because of frequent updates that make the state difficult to keep consistent without degrading the scheduling performance, even though the state itself is relatively small (about 8 bytes for round-robin information per cluster, plus another 8 bytes per landmark for the load information).

We decided to solve this problem by splitting the measurement-triggering mechanism into two parts [see Figure 3.6]. First, while responding to regular client requests, the Web servers implementing the service include *static* prefetching tags into a small fraction of their responses. Static prefetching tags do not point at any particular landmark. Instead, they point at a dedicated cluster of Web servers taking care of measurement scheduling.

The second part of the triggering mechanism is implemented by the scheduling cluster. Each machine in the cluster maintains its local scheduling state, and processes an even share of all the requests triggered by the static prefetching tags. For each such request, it invokes the scheduling policy to select the target landmark for the measurement that the request can potentially trigger.

The scheduling policy might sometimes decide not to trigger any measurement for a given prefetching request, for example when all the landmarks are overloaded. In that case, the prefetching request is serviced locally by the scheduling cluster. Note that the scheduling cluster could even exploit the fact that responding to prefetching requests is optional, and drop some of the requests completely.

Typically, however, the scheduling policy returns the address of some landmark. The scheduling cluster can then redirect the prefetching request to that land-

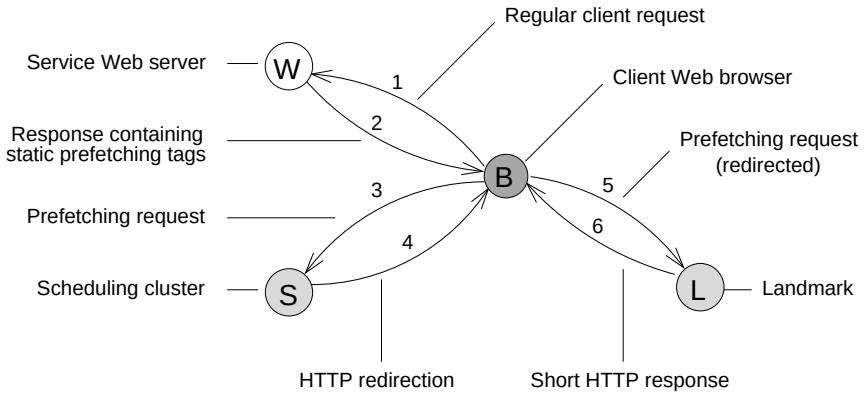


Figure 3.6: Two-phase measurement triggering

mark using an HTTP-302 response [Fielding et al., 1999]. This causes the clients to reissue the prefetching request to the landmark exactly as if the landmark address was put in the prefetching tag embedded inside the original service response. Note that although the content prefetched from the landmarks is never displayed to the users, it can still contain some brief information about the measurements being performed. This helps preventing users from becoming suspicious about the prefetching requests after they are detected by client-side firewalls.

Web server logic

Embedding static prefetching tags prevents the regular Web servers from maintaining any scheduling state, as all the prefetching tags always point at the URL of the scheduling cluster. Given that we already know how prefetching requests are processed by the scheduling cluster, the only remaining question is how to enable regular Web servers to decide whether a given response should carry a static prefetching tag resulting in a prefetching request. For now, we assume that Web servers insert at most one prefetching tag per response.

One way of enabling the Web servers to decide on insertion of prefetching tags would be to rely on client identifiers embedded in service cookies. In that case, the Web servers would include prefetching tags in responses sent to clients holding cookies with identifiers meeting the condition that:

$$ID_{client} \ \% \ X == 0$$

where X denotes some divisor value, which can be used to adjust the number of generated prefetching tags to the capacity of the scheduling cluster. An attractive property of this approach is that it keeps triggering measurements from the

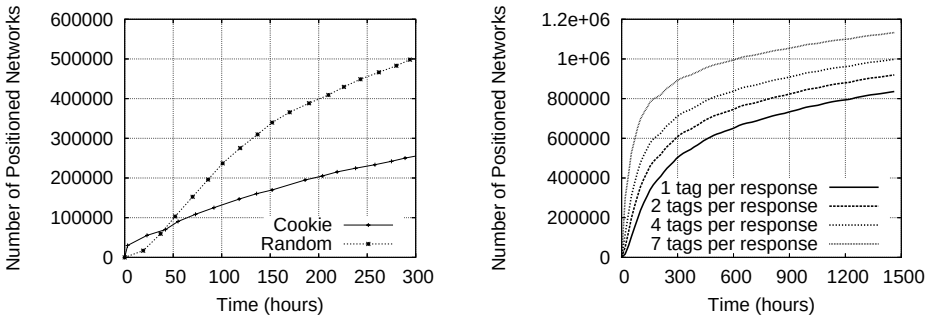


Figure 3.7: The impact of different tag-embedding strategies (a), and of different numbers of tags (b)

same clients, which should intuitively result in quickly collecting multiple measurements required to position these clients.

However, triggering measurements from the same group of clients results in only a small fraction of all the /24 networks being ultimately positioned. This can be observed in Figure 3.7(a) (the 'Cookie' line), which indicates that only about 250,000 out of the total 1.2 millions of client /24 networks were positioned after 300 hours.

The positioning coverage can be improved by inserting static prefetching tags purely at random. To this end, the Web servers include prefetching tags when:

$$random() \ \% \ X == 0$$

where $random()$ varies between 0 and some system-dependent $RAND\_MAX$ value (2^{31} on Linux), and X can again be adjusted to the capacity of the scheduling cluster. As can be observed in Figure 3.7(a) (the 'Random' line), this approach results in a larger number of /24 networks being positioned in the long run (500,000 after 300 hours), even though relying on cookies might initially seem to perform better.

While inserting static prefetching tags at random reduces the time needed to perform the latency measurements, the entire process still seems to be relatively slow. One can easily accelerate it further by inserting multiple static prefetching tags in a single service response. Clearly, inserting more tags results in triggering more measurements at the cost of increasing the load at the clients and the landmarks. This, in turn, should also lead to faster network positioning, as the fixed number of seven measurements is necessary to position each network.

Figure 3.7(b) depicts the dependency between the number of positioned networks and the number of static prefetching tags embedded in a single response. As can be observed, inserting more prefetching tags in a single response indeed

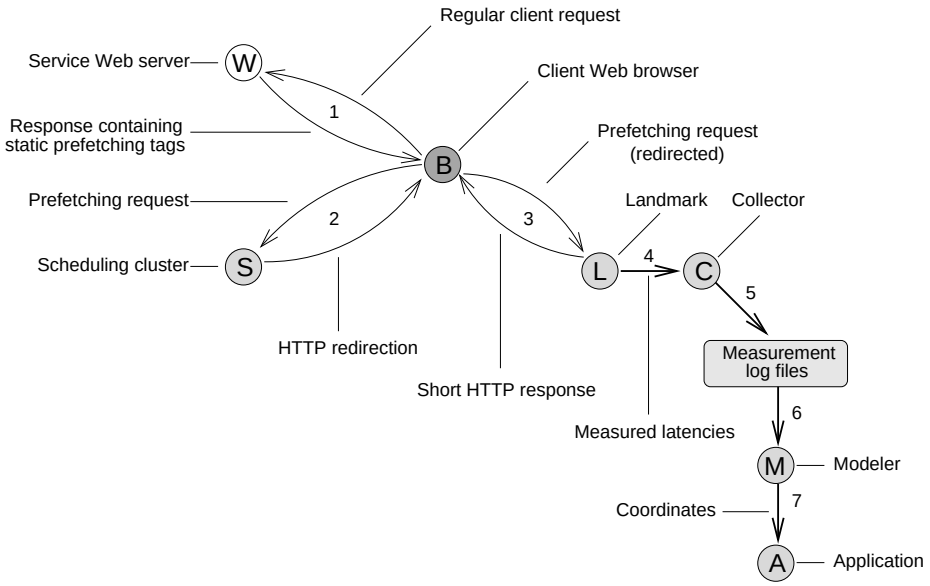


Figure 3.8: Final system architecture

helps to collect measurements faster. For example, inserting four tags per response allows for positioning more than 800,000 of /24 networks after about 450 hours, instead of 1200 hours necessary to position these networks when only one prefetching tag is embedded. Inserting seven tags per response, in turn, essentially allows for performing all the measurements necessary for positioning simultaneously, which reduces the time of positioning the 800,000+ networks to 160 hours.

3.3.4. Final architecture

The final system architecture is depicted in Figure 3.8. Latency measurements to the clients are triggered by service Web servers, which embed static prefetching tags inside a fraction of their responses.

The prefetching tags cause the clients to issue prefetching requests to the scheduling cluster, which redirects these requests to the landmarks according to the scheduling policy. This causes the clients to reissue the requests to the landmarks, which perform latency measurements while delivering a short system description. All the measured latencies are reported to the collector.

Once the latencies have been collected, they are stored in measurement logs. These logs are periodically retrieved by a special component called *modeler*, which processes the latencies and computes new sets of coordinates, as we discuss next.

3.4. LATENCY MODELING

The modeler essentially performs the two types of tasks required of any positioning implementation. First, it creates a geometric space by computing the landmark coordinates. Second, it computes all the other coordinates relative to the landmark coordinates. Both these tasks require some set of latencies as the input. It is therefore tempting to directly apply the positioning algorithm to the base latencies collected by the landmark infrastructure and stored in the measurement logs.

However, GNP requires that its input contains only one indication of latency between a given pair of machines. On the other hand, the measurement logs produced by the collector are likely to contain multiple such indications, as each landmark typically measures its latency to the same /24 network many times. Since subsequent latency measurements between the same pair of nodes are likely to return fluctuating results, the modeler must preprocess the measurement logs before their contents can be passed to the positioning algorithm.

This section describes how our GNP implementation produces coordinates based on latencies measured using the landmark infrastructure. Our implementation first isolates the typical values of pair-wise latencies from among multiple indications of such latencies found in the measurement logs. Assuming that these typical latencies remain relatively stable, they can then be used to compute coordinates remaining representative for a long time. The typical latencies are identified by means of statistical methods. As we show in our experiments, our approach results in producing coordinates that remain stable for many hours.

3.4.1. Stable latencies

In principle, latencies measured between a given landmark-node pair can fluctuate for two types of reasons. The first type are temporary intermittent conditions that do not affect long-term latencies between landmarks and nodes, such as network congestion and high CPU load. The second type are route changes, which can permanently change latencies between nodes. The goal of a good latency preprocessor is to eliminate fluctuations caused by the intermittent conditions while remaining reactive to permanent latency changes.

Clearly, network congestion can affect the observed latencies. If the path between the landmark and the node is saturated, the measurement packets are delayed by routers on the path, causing the observed latency to be longer. Note that the service should strive to reduce the impact of network congestion by avoiding it on the landmark side. This can be achieved by deploying the landmarks in hosting facilities providing hard bandwidth guarantees.

Apart from network congestion, latencies can also fluctuate because of high CPU load on either the node or the landmark. The problem with high CPU load on the node is that it might prevent the node from immediately responding to packets sent by the landmark. This can result in observed latencies being longer than they really are. On the other hand, since the packets exploited by both ICMP probing and SYNACK/ACK are handled entirely by the operating system kernels, the delay caused by high load of the node's CPU is likely to be negligible.

High load on the landmark presents a bigger problem, as it can prevent the packet sniffer running on the landmark from timestamping measurement packets accurately. The resulting inaccuracies strongly depend on sniffer implementation. We therefore assume that the observed latencies can be not only higher, but also lower than they really are.

Given that temporary intermittent conditions occur only occasionally, their resulting measurement inaccuracies can be eliminated through statistical filtering. To this end, the modeler could maintain a history of latencies measured between each landmark-node pair, and identify the real latency for that pair as the one occurring most commonly in the history. This could be achieved by means of medians, for example.

However, median latencies can change over time as well. This is caused by long-lasting conditions, such as route changes. As the route between the landmark and the node changes, its corresponding history of latencies contains more and more groups of latencies, each measured for a different route. In that case, medians calculated over complete latency histories are not guaranteed to indicate current real latencies.

We decided to detect route changes by applying the sliding percentile concept to the latency history [Pietzuch et al., 2005]. To this end, it keeps only a specific number of most recent measurements in each history, which should result in history medians being closer to the actual observed latencies.

We verified the impact of sliding percentiles on measurements used for positioning. To this end, we applied them to the latency trace collected by our system, and evaluated their performance. The trace spanned a period of six weeks and contained latencies to Google clients measured by one of our PlanetLab landmarks located in MIT. To ensure fair comparison, we analyzed latencies to only the 10,000 networks that occurred most frequently in the trace (57 times on average). The performance of sliding percentiles was evaluated by calculating the relative error between observed latencies and their corresponding values after filtering with sliding percentiles. The resulting error distribution for various configurations of sliding percentiles in depicted in Figure 3.9.

As can be observed, using sliding percentiles indeed enables one to identify current latencies more accurately, although the improvement is not very high.

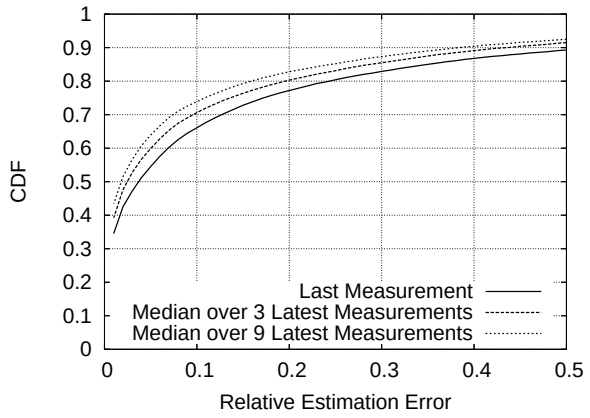


Figure 3.9: Stabilization of measured latencies

However, these small improvements result in significantly higher stability of coordinates, as we demonstrate next.

3.4.2. Stable coordinates

Computing the coordinates for a given /24 network enables one to estimate latencies between hosts in that network and those in any other network whose coordinates are already known. This allows our request redirector to identify the datacenter that is closest in terms of latency to clients in a given network, and redirect these clients accordingly.

However, latency fluctuations cause the coordinates to change over time. The degree of these changes determines how useful the coordinates are to make long-term decisions, which are important for the above applications. For example, when client requests are redirected using DNS, it can cache the responses produced by the redirecting DNS servers for several hours, which causes the redirecting decisions based on coordinates produced by our system to remain in effect for a relatively long time.

To investigate the influence of latency fluctuations on GNP coordinates, we evaluated the stability of coordinates produced by our system. We used the trace of latencies between the landmarks and the 10,000 most popular /24 networks selected for the previous experiment. We split the six-week trace into two parts. The first part was two-weeks long and was used as a basis to compute the initial coordinates of all the /24 networks. The remaining part of four weeks was used as a test trace, based on which we investigated how the coordinates of /24 networks change over time in terms of distance to their initial counterparts. To this end, for every test trace hour, we recomputed the coordinates of all the /24 networks for

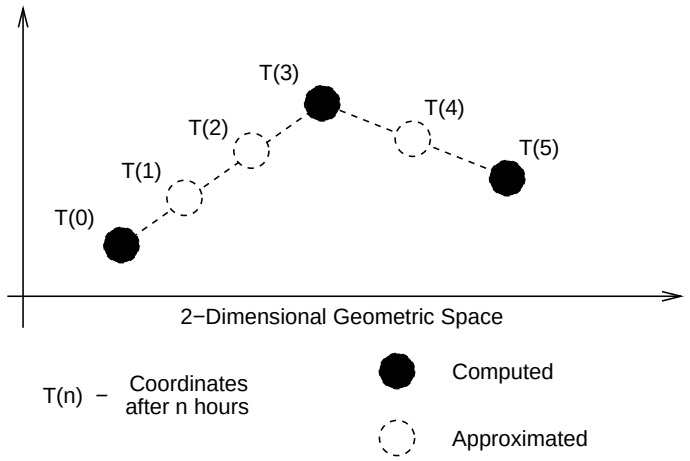


Figure 3.10: Approximating missing coordinates

which latency measurements were performed within that hour. This resulted in repositioning on average 1271 networks every hour.

Ideally, at each hour, we would compute the distance between the current and initial coordinates of each /24 network. In many cases, however, due to the lack of latency measurement within the last hour, it is impossible to compute the current coordinates directly. However, this does not mean that these coordinates did not change during that hour, but just that we did not measure latencies frequently enough.

Figure 3.10 depicts how we approximated the missing coordinates for each network. Essentially, for each pair of coordinates computed during subsequent repositioning operations, we assume that the missing coordinates between them change linearly. This enabled us to calculate the coordinates of all the networks for each test trace hour.

We evaluate the changes in coordinates during subsequent hours by calculating the median distance between the 10,000 coordinates calculated for a given hour and their initial counterparts. As shown in Figure 3.11, the coordinates change significantly when computed based on the most recent measurements (line 'Last Measurement'). They also seem to increasingly deviate from their initial values over time, as the median distance between current and initial coordinates generally increases with time. However, this hypothesis was not confirmed by a number of case studies we performed for individual networks. We therefore believe that the increasing trend is caused not by large deviation in coordinates, but by a large *number* of relatively small deviations. This number increases with time, as latencies to more and more networks become affected by route changes, leading the

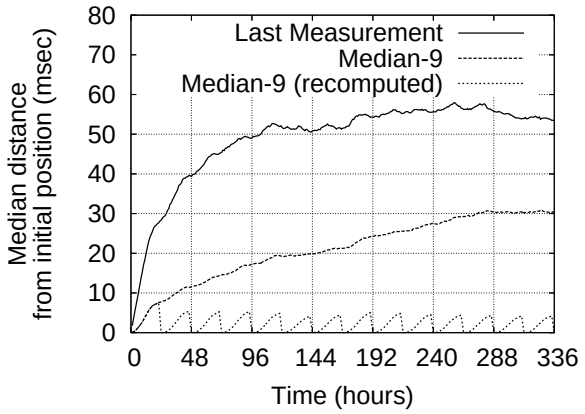


Figure 3.11: Latency stabilization vs. coordinate stability

coordinates of these networks to be significantly different. The result of aggregating such differences calculated for 10,000 networks is the increasing trend in the median distance between the current and initial coordinates.

Having observed the instability of coordinates computed based on the most recent latency measurements, we investigated whether the coordinate stability can be improved by computing coordinates based on latency measurements stabilized with sliding percentiles. To this end, we performed an experiment that was very similar to the previous one. The only difference was that the networks were repositioned based on latencies filtered using sliding percentiles. We used median latencies calculated over the set of 9 most recent measurements. The results are depicted in Figure 3.11 (line 'Median-9').

As can be observed, sliding percentiles significantly improve the stability of coordinates. However, they do not eliminate the increasing trend, which limits the maximum time for which coordinates can be relied upon. To overcome this problem, each application would need to periodically recompute coordinates so that they meet its requirements with respect to positioning accuracy. Line 'Median-9 recomputed' shows that daily recomputations can keep current coordinates within 8 milliseconds of their initial counterparts.

How often coordinates should be recomputed depends on a trade-off between the positioning accuracy and the cost of computing and propagating the coordinates to the applications. To investigate this trade-off, we ran the above experiment with *initial* coordinates recomputed every X days, for X between 1 and 7. For each of the resulting 7 simulations, we computed both the median- and the 75th percentile of distances between current coordinates and their most recently computed "initial" counterparts.

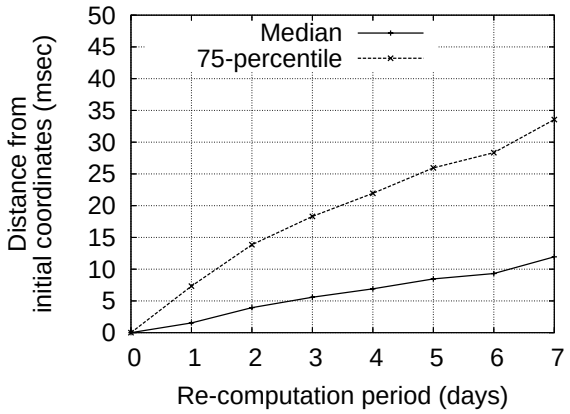
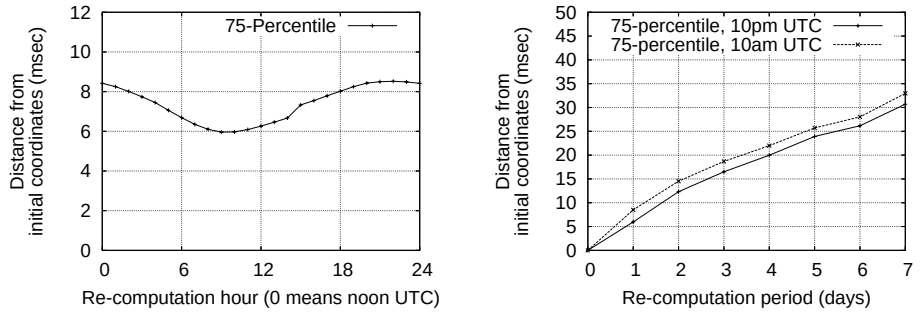


Figure 3.12: The impact of different recomputation periods

The results are presented in Figure 3.12. They indicate that daily repositioning reduces the median distance between current and initial coordinates to only 1.53 milliseconds, whereas recomputing coordinates every week results in that distance being 11.94 milliseconds. However, the corresponding 75th percentiles of distances to initial coordinates are already 7.33 milliseconds and 33.56 milliseconds, respectively. In our experiments, we decided to recompute all the coordinates on a daily basis, which, apart from offering very good stability, also makes the system very responsive to changes in network conditions.

When recomputing coordinates every day, an interesting question is whether the coordinate stability depends on the actual time of day when recomputations take place. To answer this question, we performed 24 simulations of daily repositioning based on our 4-weeks-long test trace. Each simulation was configured to recompute coordinates at a different trace hour. We evaluated the resulting stability by computing the 75th percentile of distances between current and initial coordinates observed throughout the 24 trace hours after each repositioning. The results are depicted in Figure 3.13(a).

As can be observed, the coordinates are most stable when recomputed around 10pm UTC. We believe that this is because the coordinates are then computed based on measurements collected during peak Internet hours, as they account for day time in the US and evening in Europe, which are the two continents where most of the 10,000 test networks are located. As a consequence, these coordinates remain representative for the most of the 24 hours following the recomputation, which results in 30%, or 2.5 milliseconds, improvement compared to recomputing at 10am UTC, when the stability is the worst. Note that although the absolute difference of 2.5 milliseconds might by itself seem negligible, it is easy to notice the



(a) (b)

Figure 3.13: The impact of repositioning hour on daily coordinate stability (a), and on coordinate stability in general (b)

trend in average coordinate stability, which is clearly lower when the coordinates are computed in the morning.

For the sake of completeness, we also checked how different repositioning hours influence the stability of coordinates recomputed every 2 days or more. To this end, we again simulated coordinate recomputations every N days (for N from 1 to 7) based on our test trace. We performed two simulations, each configured to recompute coordinates at a different hour: 10pm UTC and 10am UTC. The results are presented in Figure 3.13(b). As can be observed, the improvement of 2.5 milliseconds remains roughly constant irrespective of how often coordinates are recomputed, which reduces its impact to approximately 10% when recomputing coordinates every 4 days or more.

3.5. COORDINATE-BASED CLIENT REDIRECTION

Once the positioning system has been deployed, it can be used by various applications to optimize the latencies between their components. One of such applications is client redirection: based on the coordinates produced by GNP, one can redirect each client to a replica that is closest to that client in terms of latency. To this end, one first calculates the distance between the coordinates of the client and the coordinates of each replica, and then selects the replica with the shortest distance to the client. This section verifies to what extent the coordinates produced by our GNP implementation can be used for client redirection in a large-scale distributed system.

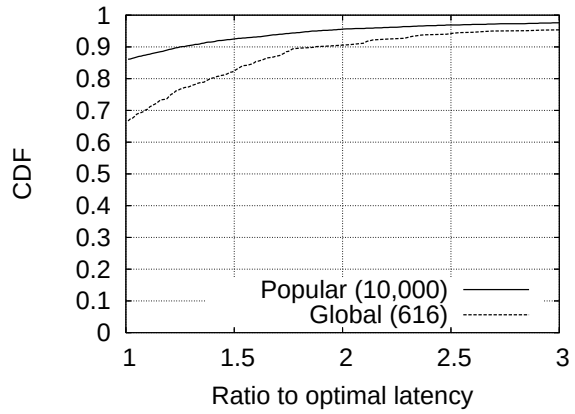


Figure 3.14: The efficiency of GNP-based replica selection

3.5.1. Absolute performance

We verified the efficiency of coordinate-based client redirection. To this end, we positioned the 10,000 /24 networks based on the median latencies measured between these networks and 20 candidate landmarks deployed on PlanetLab nodes over a period of six weeks. Each network was positioned relative to the best set of seven landmarks identified in Section 3.3.1. We chose 10 of the 13 remaining candidate landmarks such that they formed a globally distributed set of replicas. Next, for each replica, we calculated its median measured latency to each network. Finally, for each network, we determined its closest replica based on the median measured latencies, and matched that choice against that made based on latencies estimated with coordinates. The results are depicted in Figure 3.14 (line 'Popular').

As can be observed, clients from 86% of /24 networks are redirected to the replica closest to them in terms of median measured latency. Also, clients from another 10% of networks are redirected to replicas offering latencies at most two times longer than the closest ones. Finally, only about 2% of networks are redirected to replicas further than 3 times than the closest ones.

We have also performed the above experiment for the set of 616 globally distributed clients that we constructed in Section 3.3.1. The results are also depicted in Figure 3.14 (line 'Global'). It shows that coordinate-based redirection selects the closest replica for clients in about 67% of globally distributed /24 networks, and replicas offering latencies at most 2 times higher than optimal in for clients in another 24% of such networks. We believe that the suboptimal replica selection in the remaining cases is caused by node mispositioning. Nodes are typically mispositioned when they have long latencies to all the landmarks, or when the latencies

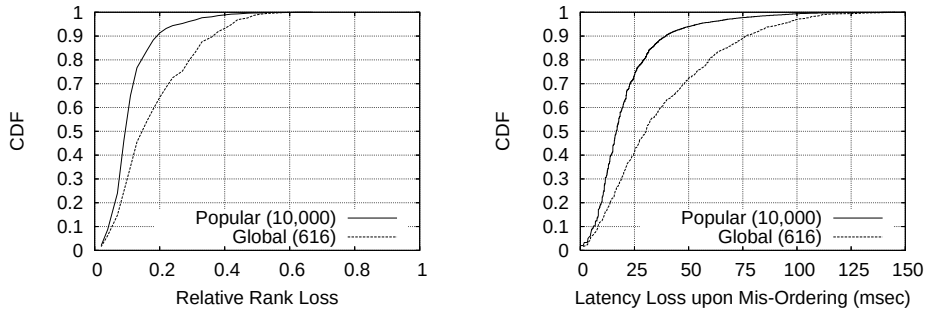


Figure 3.15: The relative performance of client redirection in terms of: relative rank loss (a), and latency (b)

of their network paths to the landmarks are self-inconsistent from the perspective of GNP, for example, because of multihoming [Zheng et al., 2005].

3.5.2. Relative performance

Although GNP-based redirection seems to perform well in terms of absolute latency values, it has recently been suggested that absolute metrics are not enough to completely evaluate redirection efficiency [Lua et al., 2005]. This is because redirected clients often care more about *relative* dependencies between latencies to different replicas, instead of their absolute values.

The relative performance of GNP-based redirection can be measured by means of another metric, called *relative rank loss (rrl)*. For each client, it creates two replica rankings: one calculated based on measured client-to-replica latencies, and another based on latency estimates provided by GNP. Given the two rankings, the *rrl* of each client C can be computed according to the following formula:

$$rrl(C,R) = \frac{\{(x,y)|\ x \neq y \text{ and } swapped(x,y)\}}{|R|(|R|-1)}$$

where R is the set of replicas, x and y are replicas, and $swapped(x,y)$ is true when the relative ordering of x and y is different in the two rankings created for client C. *rrl* can also be interpreted as the probability that $swapped(x,y)$ is true for any two different replicas.

We have computed *rrl* values based the client latencies from the test sets used to evaluate the absolute performance of client redirection. The results are presented in Figure 3.15(a).

As can be observed, *rrl* is lower than .2 for about 92% of frequent clients, and for about 65% of globally distributed clients. For these clients, any pair of replicas

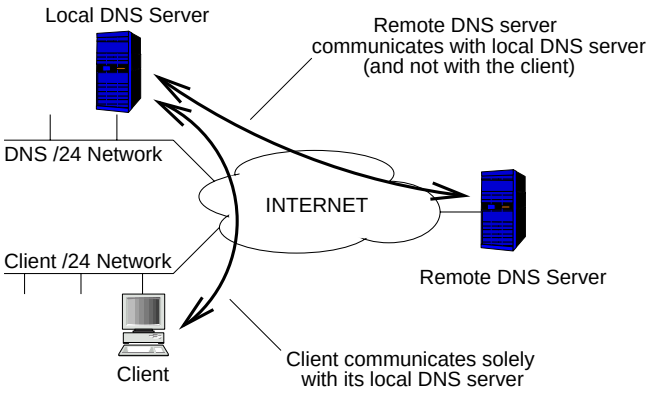


Figure 3.16: The principle of domain name resolution

has only 20% chance to be reordered when client-to-replica latencies are estimated with GNP. Furthermore, according to our data, misordering happens mostly when two replicas have very similar latencies to the client. This can be observed in Figure 3.15(b), which depicts the distribution of differences in client-to-replica latencies when a misordering occurs. It shows that loss in client-to-replica latency resulting from misordering is less than 50 milliseconds for 95% of frequent and 73% of globally distributed clients.

3.5.3. DNS considerations

A potential problem with coordinate-based client redirection is that large-scale Internet services typically redirect their clients using DNS. In that case, the redirection takes place when a client resolves the domain name of an Internet service. However, during name resolution, each such client is represented by its local DNS server, which communicates with the service DNS server on the client's behalf. The key consequence here is that the redirection policy running on the service DNS server selects replicas based on the addresses of client-side DNS servers rather than on these of the clients themselves [Shaikh et al., 2001].

The above limitation of DNS redirection is particularly troublesome when the redirecting decisions are based on coordinates produced by our positioning system. This is because these coordinates can only be computed for /24 networks that contain at least one Web client with enabled support for prefetching. Meanwhile, DNS servers are typically deployed in special /24 networks that are different from those containing regular Web clients [see Figure 3.16]. Given that our positioning system cannot determine the coordinates of most networks containing DNS servers, the redirection policy might be unable to estimate the latency between

the client-side DNS servers and the replicas, which in turn makes it impossible to select the best replica for the clients configured to use these DNS servers.

We solve this problem by associating networks containing Google clients with networks containing DNS servers these clients use. To this end, one could rely on network-aware clustering, which identifies co-located /24 networks as those falling within the same BGP prefix [Krishnamurthy and Wang, 2000]. However, this solution implicitly assumes that clients typically use DNS servers that belong to the same BGP prefix, which has been shown to be false in most cases [Mao et al., 2002]. We therefore exploit a Google proprietary mechanism that precisely discovers which DNS server is used by each Google client. The details of this mechanism are confidential and cannot be presented in this thesis.

Another problem related to DNS redirection is that client-side DNS servers may cache redirecting DNS responses of the redirector and use them to service client requests locally until these DNS responses expire. This might result in a potentially large number of clients being redirected according to old redirecting decisions, which might already be invalid, for example, because of network repositioning that has taken place in the mean time. The traditional solution to this problem is to use short-lived redirecting responses, which reduces the chance that a client-side DNS server redirects its clients according to outdated redirecting decisions [Jung et al., 2002].

3.6. GENERIC LATENCY ESTIMATION SERVICE

As our system collects latency information about millions of Internet hosts, it can potentially be used to predict latencies between arbitrary machines in the Internet, which are not necessarily Google clients. Such a generic latency estimation service could be useful for any application that needs to estimate end-to-end latencies between Internet hosts, such as a peer-to-peer overlay or a third-party content delivery network.

In this section, we investigate to what extent our system succeeds in predicting such latencies. To this end, we evaluate the accuracy of latency estimates predicted for hosts that have never been involved in any operation performed by our system. Such a non-involvement means that these hosts have never been instrumented by our system, and that we have never measured their base latencies in any way. Instead, we determine the coordinates of these hosts by simply taking the coordinates of their co-located Google clients.

A potential problem at this stage is that our system can estimate latencies only between /24 networks containing Google clients. However, while using /24 clustering allows us to position a huge number of Internet hosts, there are also

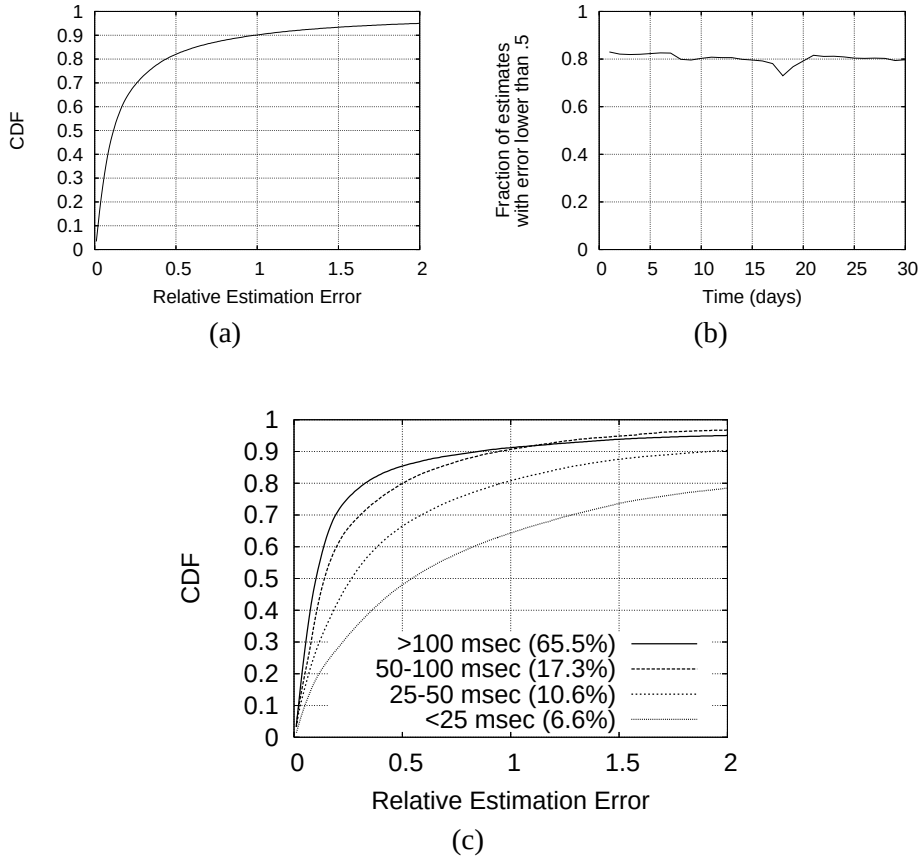


Figure 3.17: PlanetLab latency estimation: relative error (a), accuracy over time (b), and accuracy for different latency intervals (c)

many hosts that cannot be positioned when such an approach is followed. This is true for network servers, for example, which are typically deployed in different networks than user machines. We circumvent this problem by clustering Google clients into BGP prefixes, and not into /24 networks. Such coarse-grain clustering enables us to position more hosts at the expense of potential loss in estimation accuracy, as latencies to machines located in the same BGP prefixes are likely to be more diverse than those to machines located in the same /24 networks. Note that BGP clustering is only slightly slower than /24 clustering, as it can employ fast prefix matching algorithms such as that proposed by Buchsbaum et al. [2003].

Fair accuracy evaluation requires that latency estimates produced by our system are compared against their corresponding measured latencies. We use two datasets of measured latencies derived from third-party latency traces, called PlanetLab and RIPE. Both these datasets contain matrices of all-pair latencies measured between a number of machines during subsequent hours in November 2006. Each matrix is specific to a different hour and contains minimum latencies observed for given pairs of machines throughout that hour. We chose to use minimum latency values because they correspond to the "empty path" latencies that our system is striving to estimate.

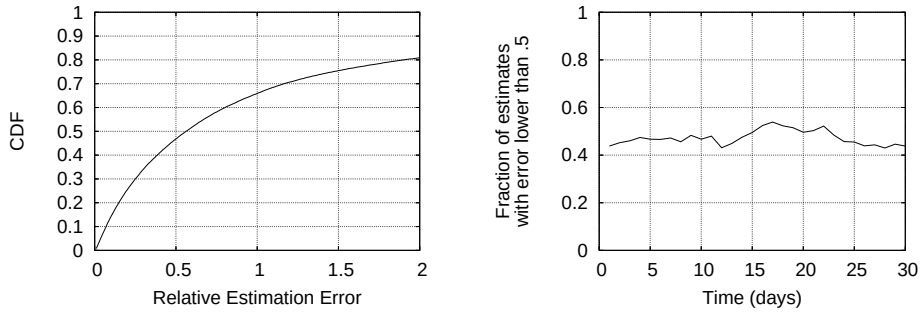
The estimation accuracy is evaluated by measuring the relative difference between latencies found in each dataset against their estimated counterparts. To ensure the fairness of comparison, all the estimates are computed based on the data collected before their corresponding measurements were performed.

3.6.1. PlanetLab latencies

The PlanetLab dataset contains latencies measured between 489 PlanetLab nodes. It was derived from the latency trace collected by Jeremy Stribling for his "all-pair pings" project [Stribling, 2006]. To this end, we aggregated the original latencies (measured every 15 minutes) into hourly all-pair matrices.

We compare the dataset latencies against their estimates provided for 327 (out of 489) PlanetLab nodes whose coordinates we were able to derive from base latencies measured to Google clients. The total number of latencies analyzed is 39.6 million (more than 50,000 per hourly matrix). For each such latency, we calculate the relative latency estimation error according to the formula introduced in Section 3.3.1. The resulting distribution of error values is depicted in Figure 3.17(a).

As can be observed, relative estimation error is lower than .5 in approximately 83% of the cases, which comes close to that reported in the original GNP paper for hosts instrumented with GNP software. More importantly, the high estimation accuracy is preserved over time. This can be observed in Figure 3.17(b), which shows how the fraction of good estimates (with error lower than .5) changes over time.



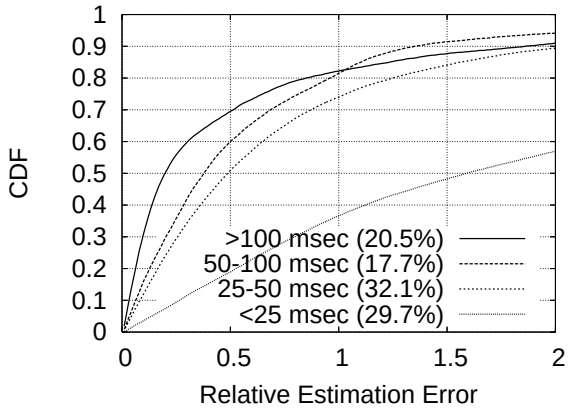


Figure 3.18: RIPE latency estimation: relative error (a), accuracy over time (b), and accuracy for different latency intervals (c)

Although the overall system performance for the PlanetLab dataset is very good, a further investigation of error values reveals that the estimation accuracy varies depending on latency magnitude. This can be observed in Figure 3.17(c), which depicts the distributions of estimation errors for four different intervals of measured latencies. The differences between these distributions indicate that precise estimation of very short latencies (25 milliseconds or less) is very difficult, as opposed to predicting long latencies (100 milliseconds or more). These results make us believe that the high overall estimation accuracy achieved for PlanetLab latencies is partially caused by their favorable distribution, as more than 65% of them are at least 100 milliseconds long.

3.6.2. RIPE latencies

The favorable properties of PlanetLab latencies are not present in our second dataset. It contains latencies measured by the infrastructure of 70 diagnostic stations deployed for the RIPE Test Traffic Measurements project (TTM) [Georgatos et al., 2001]. The diagnostic stations, called test-boxes, are deployed on the backbones of various Internet Service Providers, and used for evaluating and streamlining the communication between these backbones. Given that most of TTM ISPs are located in Europe, most of the latencies between test-boxes are very short, which makes them very difficult to estimate accurately.

We evaluate the performance of our system based on the RIPE dataset just as we did with PlanetLab. First, we use BGP clustering to position the test-boxes, which ultimately led to determining GNP coordinates for 47 of them. Given these coordinates, we calculate relative estimation errors for latencies measured between these stations, which leads to analyzing more than 1,100 measurements per hourly matrix. The resulting distribution is error values depicted in Figures 3.18(a) and 3.18(c).

As can be observed, short latencies are indeed very difficult to estimate accurately. This severely affects the overall performance, as 61.8% of RIPE latencies are shorter than 50 milliseconds. However, the accuracy of long latency estimates is far better: 70% of them are off by less than .5. Also, similar as in the case of PlanetLab, the estimation accuracy for RIPE latencies is preserved over time [see Figure 3.18(b)].

Based on the analysis performed with our two datasets, we conclude that our system could be used as a generic latency estimation service. It performs very good when estimating long latencies, which makes it particularly suitable for predicting latencies between globally distributed hosts. As for short latencies, such as those found in the RIPE dataset, they are very difficult to estimate precisely. However, our system can still estimate at least some of them with a reasonable degree of accuracy.

3.7. FEDERATED LATENCY ESTIMATION

Whereas our generic latency estimation service can be useful for many external systems, there are still cases when distributed systems might prefer to estimate latencies by themselves. A good example here is a system whose operation relies on latency estimates, but whose components are located in the parts of the Internet that are not covered by our latency estimation service. Another example if a system that wants to remain autonomous and independent of any external services.

Systems unwilling to use our latency estimation service could simply deploy their own ones. However, as our design relies on a number of stable and high-performance components, such an approach might turn out to be infeasible in decentralized systems running on unreliable or low-performance hosts. Instead, such system need a solution that minimizes dependencies between hosts, for example, by preserving the system decentralization.

This section discusses how to estimate latencies in large-scale federated environments by allowing each host in such an environment to run its own instance of a positioning system. Our approach allows each host to select its own positioning parameters such as the space dimension, the selection of landmarks and the positioning algorithm. This enables each host to adjust the parameters to its own capacity and requirements with respect to positioning accuracy. More importantly, maintaining independent private spaces enables federated hosts to autonomously reconfigure their private spaces if necessary, thus relaxing inter-host dependencies even further. In fact, the only bounds between federated hosts are caused by the fact that landmarks are usually selected from among other federated hosts, which effectively leads to turning small groups of hosts into autonomous measurement infrastructures.

An important property of private spaces is that they preserve estimation correctness: even though multiple hosts in a federated system may estimate latencies in private spaces with different parameters, the resulting estimates are highly correlated among all the spaces. This allows the federated system to ignore the fact that estimates are produced autonomously at different locations, and consider all the independent instances of a positioning system as a single latency prediction service. As a consequence, the federated system might run distributed algorithms relying on consistent latency information in a completely decentralized manner.

3.7.1. Private spaces

When positioning hosts in a federated environment, a crucial observation is that in most cases hosts are interested in latency estimates rather than in host coordinates themselves. We conclude that it is not necessary that all hosts use the same global space definition. Instead, we propose that each host interested in latency estimation runs an independent instance of the positioning process.

The architecture of each instance resembles that of centralized positioning system 3.19. Each ('drafter') host running its own instance employs a number of ('helper') hosts from among other federated hosts to act as landmarks measuring their latencies to ('target') hosts being positioned. The number and choice of helpers are autonomous decisions of each drafter. However, we assume that each helper is able to perform latency measurements according to instructions from the drafter, potentially using techniques proposed for the centralized version in

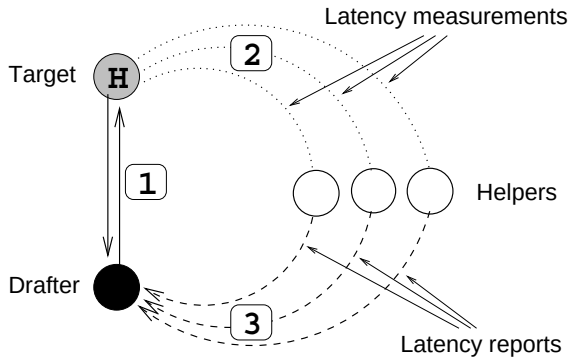


Figure 3.19: The architecture of a single GNP instance in a federated environment

Section 3.3.2. Note that the distinction between drafter, helper, and target hosts only refers to their roles in a single instance. A given host may at the same time be the drafter of its own space, a helper to some other drafters, and a target for yet other drafters.

Based on measurements provided by helpers, each drafter constructs its own private geometric space. To assign landmark coordinates, each drafter instructs its helpers to measure and report latencies among each other. After a drafter has assigned the landmark coordinates, it can position targets in its private space based on the latencies between the landmarks and the targets. It may also decide to reposition them some time later, if it notices that their coordinates are no longer accurate. In both cases, the drafter is the only one to decide whether and how (re)positioning should take place. This decision depends solely on the drafter configuration and can be taken independently from other drafters and from the targets themselves.

As the operation of each instance closely resembles that of our centralized latency estimation system, each host running such an instance needs to perform several tasks by itself, including measurement scheduling, latency collection, and coordinate computation. We assume that the number of positioning operations in each private space is low enough to allow each host to handle all these tasks without becoming overloaded.

3.7.2. Discussion

By allowing each drafter to independently construct its private geometric space and to position targets in that space, we enable each drafter to adjust the latency estimation process to its own requirements. Adjustments may include the selection of the space dimension, the set of helpers, and the positioning algorithm.

Another consequence of using private spaces is complete decentralization. It eliminates the need for global agreement and configuration, since there is no single configuration aspect on which all the hosts must agree or to which they must obey. Although running multiple instances requires that helpers cooperate with their drafters, these cooperating groups are small and independent from each other. This property makes our approach particularly suitable for large-scale distributed systems, where neither global knowledge nor coordination is possible.

Positioning hosts in private spaces poses a difficulty when drafters need to send host coordinates to each other. Such coordinate transfers can be useful in certain situations, for example to request information about hosts located within a certain radius from a given location. This problem can be addressed by representing a target's location not as a set of absolute coordinates, but as a set of latencies between that target and a (potentially large) number of reference hosts. Provided that the receiver has already positioned a significant subset of the reference hosts, it will be able to derive the coordinates of the transferred location within its own private space. Because the selection of the reference hosts can be negotiated, and because the latencies can be in fact latency estimates, any two drafters are still able to pass coordinates to each other.

In contrast to coordinate passing, exchanging of raw latency measurements is relatively straightforward. A good reason for drafters to exchange such measurements is updating or providing additional latency information. In a large-scale system, it is likely that some targets contact more than one drafter during their lifetime. In this case, several drafters would be interested in latencies to such targets. Such drafters might therefore exchange recent latency information about these targets in order to provide better latency estimates.

There are several techniques enabling efficient data sharing in large-scale systems, such as lazy dissemination and publish-subscribe systems. In general, they ensure that the data sent by one host are eventually propagated to other hosts interested in the data. Note that both these properties fit into the concept of federated latency estimation, as it requires neither immediate data propagation, nor the guarantees that all drafters receive the data.

An important observation is that sharing raw latency measurements does not force any drafter to use them. In fact, each drafter retains the option of deciding which measurements should be used for target positioning. One reason not to use all the measurements is security. In principle, each drafter should only rely on measurements that originate from helpers it trusts. This can be facilitated by instructing each helper to sign its produced measurement results such that each drafter can identify the measurement origin and decide whether it should be trusted or not.

3.7.3. Evaluation

Drafters estimate latencies independently from each other. However, if multiple drafters jointly run the same application, it is desirable that latencies estimated by different drafters are highly correlated. For example, if an application requires each drafter to build a minimal host-spanning tree based on latency estimations, then estimation inconsistencies can result in each drafter building a different tree, which can lead to a malfunctioning application. On the other hand, if the correlation coefficient is high, then the application may rely on local estimations, and consider the rare cases when estimations are inconsistent as exceptions.

In this section, we show that latencies estimated in different private spaces are consistent with each other. We start with describing the datasets we use in our experiments, and discussing our experience with selecting the most appropriate space dimension. Then, we evaluate the correlation of spaces using the same positioning algorithm. Finally, we investigate the impact of using different positioning algorithms on estimation accuracy, and measure the correlation of spaces using different positioning algorithms. All the experiments are performed in 6-dimensional geometric spaces, which has been shown to offer highest positioning accuracy [Szymaniak et al., 2004].

Dataset description

We evaluate our approach based on three independent datasets, each containing a snapshot of all-pair latencies measured among a set of machines in the Internet. The first dataset was collected on June 25, 2003, using King [Gummadi et al., 2002]. This tool allows to estimate latencies between any pair of DNS servers, provided that they support recursive DNS queries. We chose 100 DNS servers such that they were diversified in terms of both geographical location and IP address prefix. In particular, each of them belonged to a different Autonomous System. We measured the latencies between each pair of the DNS servers. As it turned out, 37 of them did not support recursive DNS queries, which left us with a snapshot of latencies among the remaining 63 DNS servers. Note that fair distribution of the DNS servers over the entire Internet may result in an abnormally low occurrence of short latencies.

The second dataset was collected on October 1, 2003, using the RIPE-NCC's Test-Traffic Measurements Service [Georgatos et al., 2001]. The service infrastructure consists of a number of probing hosts deployed on the backbones of Internet Service Providers. We selected 40 of the probing hosts such that the round-trip time between any pair of them was at least 1 millisecond. The resulting set contained 33 probing hosts in Europe, 5 in the USA, 1 in Asia, and 1 in Australia.

Since most of the probing hosts are located in the same continent, this trace may be biased toward short latencies.

The third dataset was collected on November 19, 2003 on 60 nodes within PlanetLab [Bavier et al., 2004]. PlanetLab is a distributed infrastructure of Linux hosts deployed in academic and research institutions. The hosts we used were located mainly in the USA (46), but also in Europe (7), Asia (5), Australia (1), and South America (1). We measured round-trip times between each pair of hosts using the SYNACK/ACK method [Andrews et al., 2002].

Helper selection versus space correlation

Different drafters construct their private spaces independently based on different sets of measurements. In this experiment, we investigate to what extent the latency estimations derived from different private spaces are consistent with each other. To do this, we create two private spaces based on disjoint sets of landmarks. All other hosts are positioned relative to the landmarks. For each pair of hosts, we compare the two latencies estimated within the two private spaces.

Figures 3.20(a), 3.20(b), and 3.20(c) plot the latencies estimated within two private spaces chosen at random and constructed from the King, RIPE, and PlanetLab datasets, respectively. Each point (x,y) corresponds to the latencies x and y estimated in the two private spaces for the same pair of hosts. As can be observed, most points are close to the ideal line $y = x$.

However, the correlation of two private spaces obviously depends on the choice of these spaces. We formally express the correlation between two given spaces by means of the classical C_{cor} correlation coefficient [Edwards, 1976]. It denotes the quality of a least squares fitting between estimates made in different spaces, and varies between 1 (identical spaces) and 0 (no correlation). In order to obtain meaningful results, we calculated correlation coefficients for each possible pair of 100 random spaces. As it turned out, there were pairs of spaces for which C_{cor} was close to 0. Detailed analysis of these pairs revealed that they contained at least one *self-inconsistent* space.

Space self-inconsistency is a phenomenon that occurs when a poor choice of landmarks causes the *inter-landmark* latencies to differ significantly from their corresponding Euclidean distances. Since the Internet is not a perfect geometric space, some discrepancies between inter-landmark latencies and their corresponding Euclidean distances can always be observed, and GNP strives to minimize these discrepancies when calculating landmark coordinates. Nevertheless, for some landmark sets, large discrepancies cannot be eliminated by the algorithm calculating landmark coordinates, causing these landmark sets unsuitable for accurate latency estimation.

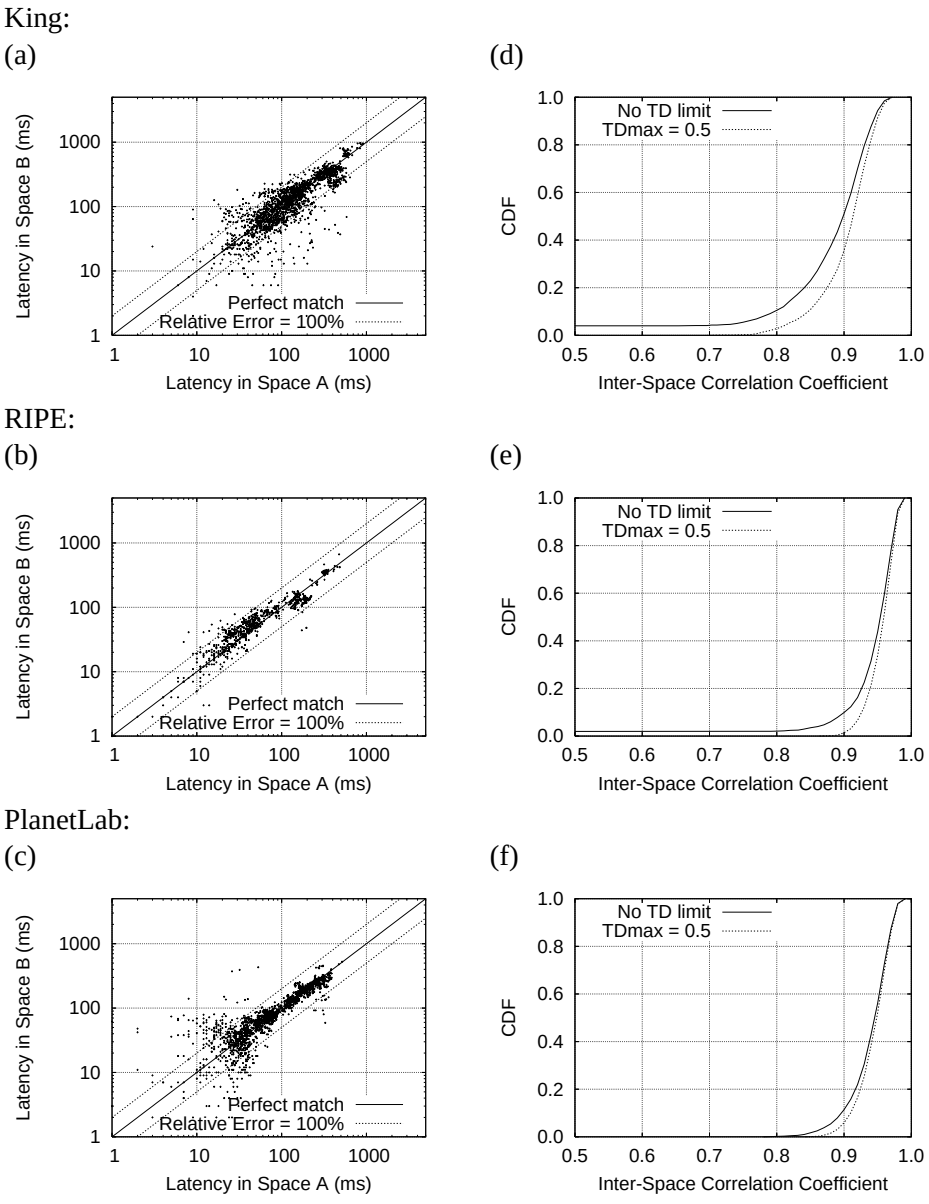


Figure 3.20: Helper selection versus space correlation: two independent spaces and 1000-space pair CDFs

Maximum *TD*		∞	2	1	0.5	0.25	0.1
King	Eliminated spaces (%)	0	4	10	21	39	72
	Average C_{cor}	0.85	0.89	0.89	0.90	0.91	0.92
RIPE	Eliminated spaces (%)	0	3	6	14	27	57
	Average C_{cor}	0.93	0.95	0.95	0.96	0.96	0.96
PlanetLab	Eliminated spaces (%)	0	1	2	5	14	35
	Average C_{cor}	0.94	0.94	0.94	0.94	0.95	0.95

Table 3.1: Improving space correlation by restricting the *TD* value

In order to detect self-inconsistent spaces, we defined the total discrepancy *TD* for a given space as the sum of all latency-vs-distance discrepancies observed for individual landmark pairs:

$$TD(L_1,\ldots,L_k) = \sum_{i\geq 1}^{k}\sum_{j>i}^{k} \varepsilon(d_{L_iL_j}, d^*_{L_iL_j})$$

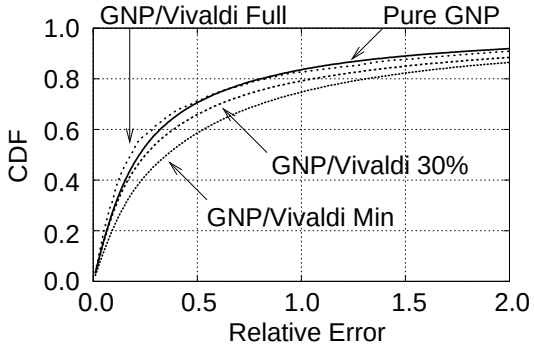
where $d_{L_iL_j}$ and $d^*_{L_iL_j}$ respectively denote the measured latency between hosts L_i and L_j and the corresponding Euclidean distance, and $\varepsilon(.)$ denotes a classical error function:

$$\varepsilon(d_{L_iL_j}, d^*_{L_iL_j}) = \left(d_{L_iL_j} - d^*_{L_iL_j}\right)^2$$

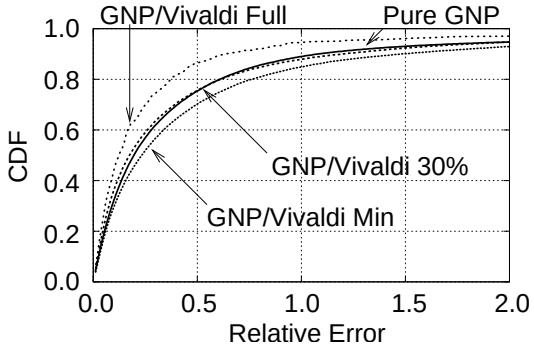
Note that the higher value of *TD*, the more self-inconsistent the space for which *TD* has been calculated. Since using self-inconsistent spaces degrades the space correlation, we decided to eliminate them by restricting the maximum *TD* of private spaces that can be used by drafters. In a real-life system, landmark sets producing self-inconsistent spaces can be easily identified, in which case a drafter should choose a different landmark set instead.

Figures 3.20(d), 3.20(e), and 3.20(f) depict the cumulative distributions of correlation coefficients for TD_{max} equal to 0.5 and ∞ calculated for the King, RIPE, and PlanetLab datasets, respectively. For each TD_{max}, we used the same set of 100 random spaces, but we filtered out spaces where $TD > TD_{max}$. Then, we calculated correlation coefficients for all pairs of the remaining spaces. The statistics for this experiment are presented in Table 3.1, which also includes results for TD_{max} equal to 2, 1, 0.25, and 0.1. As can be observed, using $TD_{max} = 0.5$ eliminates 5% to 21% of pairs (depending on the dataset), but leads to average C_{cor} of at least 0.9. Interestingly, although limiting *TD* seems to be important for inter-space correlation, it only slightly improves the estimation accuracy in general. We plan to investigate this phenomenon in the future.

King:



RIPE:



PlanetLab:

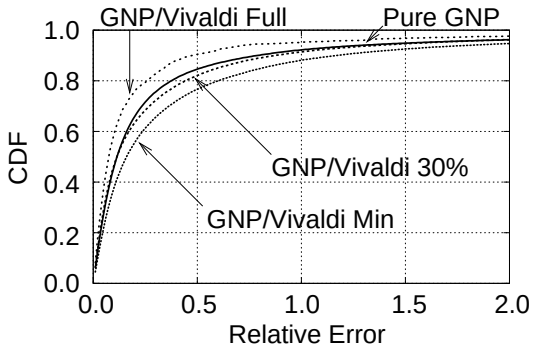


Figure 3.21: Algorithm selection versus estimation accuracy: error distribution

Algorithm selection versus estimation accuracy

Each drafter positions its targets based on the measured latencies between the targets and its chosen helpers. In addition, latencies must also be measured between each pair of helpers to assign coordinates to each helper in the space construction phase. Still, however, the latency estimations are derived only from a small subset of all potential measurements that could (theoretically) be performed for each pair of hosts in the system. An interesting question is whether and to what extent the latency estimations would improve if each drafter measured the latencies between each target pair, and then positioned all the targets based on a full set of measurements between them, using some global optimization algorithm. In other words, we compare the accuracy of GNP to the theoretical best accuracy among latency estimation techniques based on host positioning.

To calculate the maximal accuracy, we applied the global optimization scheme proposed by Vivaldi [Cox et al., 2004] to each of our datasets, every time using *all* the latencies they contain. Then, we compared the thus-obtained optimal accuracy to that of pure GNP.

The accuracy comparison is presented in Figure 3.21 (lines labeled "Pure GNP" and "GNP/Vivaldi Full"). As can be observed, running GNP/Vivaldi Full on the King dataset did not result in any improvement compared to the results obtained with Pure GNP. On the other hand, the accuracy increased for the other two datasets. Interestingly, the improvement was clearly higher for the RIPE dataset than for that collected on PlanetLab. Since Pure GNP and GNP/Vivaldi Full produce similar (relative) host coordinates (as we show in Section 3.7.3), we believe that the difference in estimation accuracy is caused by *slight* differences between relative coordinates themselves. These differences have almost no impact on the estimation of long latencies constituting the main part of the King dataset, but they do affect the estimations of short latencies enclosed in the RIPE dataset. This phenomenon also explains why the diversity of latencies in the PlanetLab dataset causes GNP/Vivaldi Full to only moderately improve the estimation accuracy.

As collecting a full set of latencies is usually not possible, a question arises how good would the global optimization approach perform on a minimal data subset required by Pure GNP. Such a minimal subset consists of latencies between each of $N+1$ helpers and all the other hosts. To answer this question, we applied Vivaldi to 100 minimal data subsets, each including a random selection of helpers.

The results are presented in Figure 3.21 (lines "Pure GNP" and "GNP/Vivaldi Min"). For all three datasets, GNP/Vivaldi Min performed worse than Pure GNP. This is not surprising, as Vivaldi assumes fair distribution of measurements across all the hosts, and not their concentration on a small number of helpers.

Since GNP/Vivaldi Min performed worse than Pure GNP, we wondered how many helpers are necessary for a drafter running Vivaldi to achieve the same per-

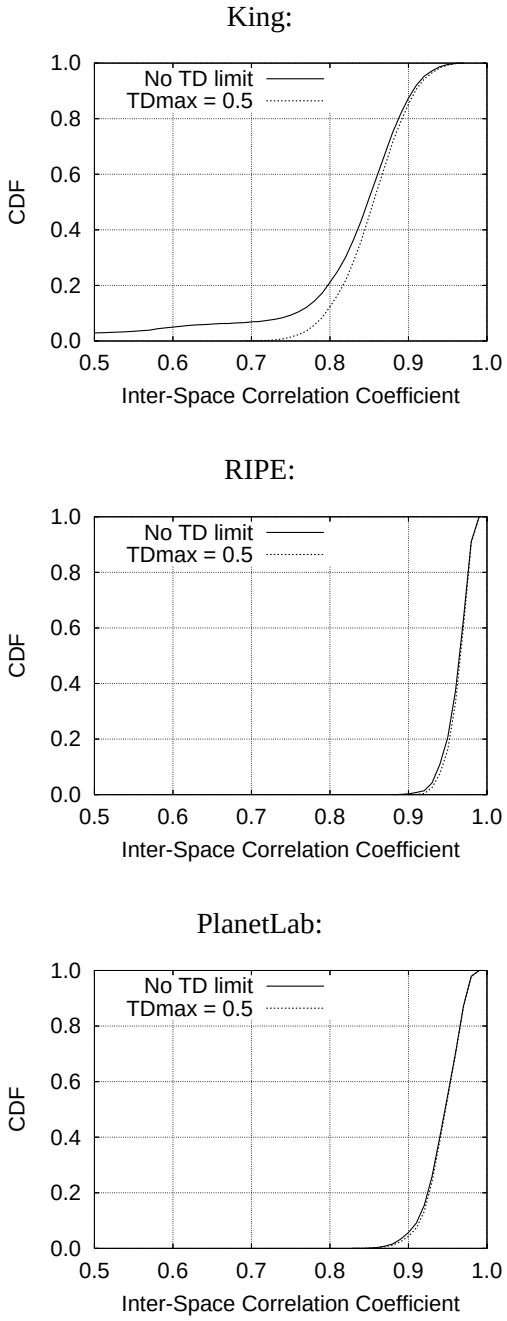


Figure 3.22: The correlation between Pure GNP spaces and GNP/Vivaldi 30% spaces

formance as drafters using the Pure GNP scheme. In order to find out, we again applied Vivaldi to 100 random data subsets. This time, however, instead of containing the minimal number of $N+1$ helpers, they included from 20% to 100% of the hosts as helpers.

The aggregated results for this experiment are shown in Figure 3.21. Except for the King dataset, the accuracy of GNP/Vivaldi is similar to that of Pure GNP, if the number of helpers included in a data subset is about 30%. This result could not be observed for the King dataset, as it did not outperform Pure GNP even when the entire dataset was used. We conclude that to benefit from running Vivaldi, a drafter would have to use more hosts as its helpers. Although it is unlikely for a drafter to have a large number of helpers, there are methods of obtaining measurements from non-helper hosts via measurement sharing, as we mention in Section 3.7.2. Also, note that the accuracy achieved by Pure GNP using minimal datasets is not much worse than the theoretical optimum achieved by GNP/Vivaldi Full. This indicates that gathering more data than really necessary may turn out be expensive compared to the expected gain in estimation accuracy.

Algorithm selection versus space correlation

Except for choosing its own landmark set, each drafter can decide on which positioning algorithm to use. In this section, we discuss how this decision affects the correlation of latency estimations by comparing different positioning algorithms.

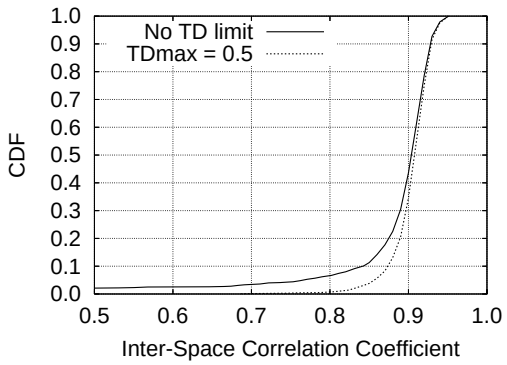
We applied both GNP/Vivaldi and Pure GNP to 100 random data subsets of each of our datasets. We ensured that both achieve similar *accuracy* by using minimal data subsets for Pure GNP, and 30% data subsets for GNP/Vivaldi (see Section 3.7.3). Then, we calculated the correlation coefficients for all space pairs, where one is made with Pure GNP, and the other with GNP/Vivaldi 30%.

The results are presented in Figure 3.22. In all datasets, the spaces generated by Pure GNP are highly correlated with these generated by GNP/Vivaldi 30%. However, since there is some difference in accuracy achieved by GNP/Vivaldi 30% and GNP/Vivaldi Full, we decided to compare the correlation of Pure GNP spaces against the latter as well.

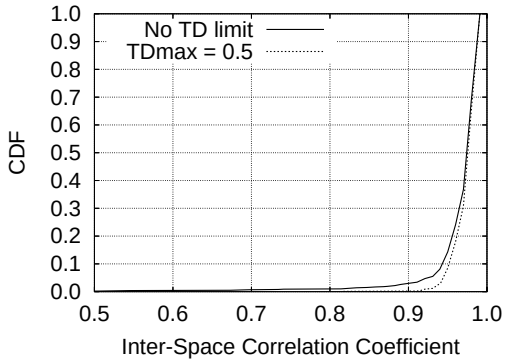
The results are presented in Figure 3.23. Also in this case, estimations made by Pure GNP using minimal data subsets are highly correlated with the theoretical optimum achieved by GNP/Vivaldi Full.

These three experiments prove that, even if drafters use different algorithms to construct their private spaces, their latency estimates remain consistent. Moreover, the estimates are also consistent with the theoretical optimum. This property allows drafters to perform their estimations independently, even if they jointly run some distributed application that requires consistent estimates to operate correctly.

King:



RIPE:



PlanetLab:

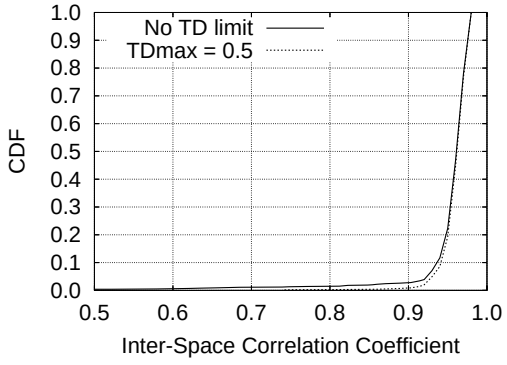


Figure 3.23: The correlation between Pure GNP spaces and the GNP/Vivaldi Full space

3.8. CONCLUSION

We have presented an implementation of GNP incorporated into the Google content delivery network. In contrast to its previous counterparts, our implementation does not rely on active participation of Web clients, as all the latency measurements are performed passively by the landmarks. The overhead incurred by these measurements is carefully controlled by a scalable centralized scheduler, which prevents both the landmarks and the network from becoming overloaded. Deploying our solution requires only a small number of CDN modifications, which makes it attractive for any CDN interested in large-scale latency estimation.

Our system has collected latency information about millions of Google clients for several months. The analysis of these data confirmed many results presented in earlier research on GNP. We add to these results by investigating the issue of coordinate stability over time. We have shown that coordinates drift away from their initial values with time, making 25% of the coordinates to become inaccurate by more than 33 milliseconds after one week. However, daily recomputations make 75% of the coordinates stay within 6 milliseconds of their initial values.

Apart from analyzing the behavior of GNP coordinates over time, we have also discussed our experience with their practical applicability. We have demonstrated that coordinate-based redirection policy selects replicas closest to the clients in term of *measured* latency in 86% of all cases. In another 10% of all cases, the selected replicas offer latencies at most two times longer than optimal.

Collecting a huge volume of latency data has enabled us to estimate latencies between globally distributed Internet hosts that have not participated in our measurements. We have been able to determine the coordinates of such hosts by applying network-aware clustering. The results are sufficiently promising that Google may offer a public interface to the latency estimates in the future. Such an interface could be useful for any large-scale distributed applications, including peer-to-peer overlays and other content delivery networks.

Finally, we have demonstrated that positioning-based latency estimation can also be implemented in federated environments. In that case, each host constructs its private geometric space and positions other hosts in that space. The private space parameters and the positioning algorithm can be adjusted on a per-host basis. More importantly, since private spaces are mutually independent, positioning operations performed at different hosts do not need to be coordinated, which makes the system more scalable. At the same time, latency estimates performed in different private spaces are highly correlated with each other, which allows one to run algorithms relying on latency estimates in a completely decentralized manner. A good example of such an algorithm is HotZone – our replica placement algorithm that we discuss in the next chapter.

CHAPTER 4

Replica Placement

4.1. INTRODUCTION

Replication is commonly employed by modern Internet services to improve the communication delay experienced by their clients. Such services typically deploy their several replicas in different parts of the Internet and automatically redirect each client to its proximal replica [Dilley et al., 2002]. Doing that can significantly improve the communication delay between the client and the service, resulting in a better client experience.

When replicating an Internet service, an important issue that must be addressed is where to place replicas. Different replica placements are likely to result in different client-experienced communication delay, hence this decision is crucial for optimizing the service response time.

Several algorithms have been proposed to address the replica placement problem [Karlsson et al., 2002]. The Greedy algorithm, for example, places replicas one-by-one, each time exhaustively evaluating all the possible replica locations. It has been shown to produce very good placements, yet its computational cost is quite large: $O(K \cdot N^2)$, where K is the number of replicas, and N is the number of potential replica locations. Another algorithm, called HotSpot, places replicas on nodes that along with their neighbors generate the greatest load. It has a slightly lower computational cost: $O(N^2 + min(N \cdot logN + K \cdot N))$. On the other hand, its produced placements are not as good as those of Greedy. The high computational cost of these two algorithms prevents them from scaling for systems with more than 10^4 potential replica locations [Karlsson et al., 2002]. This bound is unacceptable for current worldwide replicated systems, which often need to consider at least 10^5 of such locations, and in general for the freely-scalable systems of the future [Dilley et al., 2002].

We propose to reduce the time needed for placement computation by taking a two-step approach. The first step is to select *network regions* where replicas should be placed. A network region is a group of nodes whose latencies to each other are relatively low. Other properties of nodes, such as the available storage space, network connection bandwidth, and availability, are irrelevant at this stage. This simplification is likely to accelerate the corresponding region-selection algorithm, especially if the latencies are modeled efficiently.

Once the network regions have been identified, the second step is to choose individual replica-holding nodes in different regions. This time, however, it is possible to consider all the metrics ignored during the first step, as the number of nodes in a region is much smaller. Since these metrics tend to be node-specific, we believe that it is not necessary to consider them at the global level.

Our two-step approach clearly prioritizes client-to-replica latency over the other metrics. This is the direct consequence of our considerations in Chapter 1, where we observed that latencies tend to influence not only the communication delay but also the bandwidth available to client TCP connections. Believing that network latencies are the key factor affecting the system response time, we first focus on latency optimization, and deal with other metrics later.

This chapter discusses HotZone, an algorithm that addresses the first step, that is the identification of the regions where replicas should be placed. For evaluation purposes, we assume that the second-step algorithm simply selects the node with minimal average distance to all other nodes in the region.

HotZone relies on the fact that Internet latencies can be modeled in an M-dimensional geometric space, as discussed in the previous chapter. In this model, nodes are assigned M-dimensional coordinates. The latencies between any two nodes are modeled as the distance between their corresponding coordinates. HotZone identifies network regions as groups of nodes with proximal coordinates and places replicas in the most active regions. In this way, it avoids the costly pairwise latency estimations between all potential replica locations. This reduces the computational cost of HotZone to $O(N \cdot max(logN, K))$, which is significantly lower than those of previously proposed algorithms.

We compare HotZone with four other placement algorithms, including Greedy and HotSpot. We show that the placement quality offered by HotZone is on average off by 5% from that of Greedy. We also show that HotSpot does not produce satisfactory results when directly applied to node coordinate sets, and discuss how a simple modification can make this algorithm achieve the performance close to that of HotZone, although the computation time remains unsatisfactory. Finally, we demonstrate that the lower complexity of HotZone leads to significant gains in placement computation times, up to 3 orders of magnitude when placing 20 replicas.

The rest of this chapter is structured as follows. Section 4.2 discusses relevant research efforts. Section 4.3 describes the details of our replica placement algorithm, and analyzes its computational complexity. Section 4.4 evaluates the algorithm performance. Finally, Section 4.5 concludes.

4.2. BACKGROUND

Many replica placement algorithms have been proposed by the research community [Karlsson et al., 2002]. In general, their goal is to either optimize client performance given an existing infrastructure, or minimize the infrastructure cost while achieving given client performance at the same time [Karlsson and Mahalingam, 2002]. This performance goal can be expressed by means of many different metrics, but placement algorithms typically optimize only one of them: client-to-replica latency. As discussed in previous chapters, minimizing network latencies between clients and replicas plays a key role when optimizing the response time of replicated Internet services.

Several placement algorithms try to optimize the client-to-replica latency by minimizing the hop count between clients and replicas [Radoslavov et al., 2001; Jamin et al., 2001; Tang and Xu, 2004]. The underlying assumption is that the latency of a network path mainly depends on the number of individual links that this path consists of. Optimizing hop counts is attractive, because they are relatively stable and easy to measure [Paxson, 1997b]. On the other hand, the accuracy of hop-based latency estimation is relatively poor [Huffaker et al., 2002].

Since the replica placement problem is NP-complete, placement algorithms optimize their chosen metrics by means of heuristics [Karlsson and Karamanolis, 2004]. An example of such a heuristic is the Greedy algorithm, which determines replica locations one-by-one [Qiu et al., 2001]. It starts with exhaustively evaluating all possible locations for the first replica, and then choosing the location yielding the best performance. The subsequent replicas are placed in the same manner, except that starting from the second replica, all the previously placed replicas are considered to be fixed during metric calculation. Greedy can be used to optimize any metric. When optimizing the hop count metric, the Greedy algorithm produces placements that have been shown to be within a factor of 1.5 of those yielding optimal client performance.

The computational complexity of the Greedy algorithm is $O(K \cdot N^2)$, where K is the number of replicas to be placed and N is the number of potential replica locations. In a large-scale distributed system, N can be very large, leading to long-lasting computations. In such cases, the value of N can be reduced by clustering [Krishnamurthy and Wang, 2000]. However, even relatively coarse-grained

clustering may be unable to reduce N to a value for which the Greedy algorithm can be run efficiently. For example, clustering based on Autonomous Systems typically groups nodes into a few thousands of Autonomous Systems, which means that the Greedy algorithm still has to perform millions of latency estimations to place a single replica.

Another heuristic, called *HotSpot*, places replicas on nodes that generate the greatest load [Qiu et al., 2001]. It first orders the nodes according to the amount of traffic generated jointly by each node and its neighboring nodes. Then, it places replicas on the first K nodes with the most active neighbors. Being a neighbor node is defined in terms of some network distance metric. For example, a node can consider all nodes whose latency is less than X milliseconds to be its neighbors. The placements produced by the HotSpot algorithm have been shown to be within a factor of 1.6 – 2.0 of the optimal one when optimizing on the hop count metric. Compared to the Greedy algorithm, the drop in efficiency is traded for only a slightly more attractive complexity, which is $O(N^2 + min(N \cdot logN + N \cdot K))$. This can still be too much for globally distributed systems.

Since several heuristics are capable of producing a placement decision, it may be desirable to dynamically select the one that produces the best results for a given system topology and workload [Karlsson and Karamanolis, 2004]. A naive solution could be to choose the best heuristic after simulating all the possible ones. This is often infeasible, as such a simulation is costly in terms of time and computing resources. Additionally, heuristic selection may have to be performed every time the system workload changes.

A promising alternative for costly heuristic simulations is model-based heuristic analysis [Karlsson and Karamanolis, 2004]. The authors propose to model each placement heuristic in terms of an integer programming formulation of the replica placement problem. It is then possible to calculate an upper bound for the performance offered by each heuristic given a system topology, a workload, and a performance goal. Assuming that the actual performance of a heuristic is close to the upper bound, one may select the best heuristic as the one with the highest upper bound.

The model-based heuristic analysis may accelerate the process of heuristic selection by saving the time otherwise needed for heuristic simulations. However, it is still required that all the heuristics are modeled and exhaustively evaluated, which can still be too costly, especially in the case of large systems. Moreover, if the number of constraints that must be considered by an algorithm is low, we may be able to identify a heuristic that invariably produces near-optimal results. In that case, the two phases required by the model-based analysis may simply turn out to be redundant.

4.3. ALGORITHM

The goal of a replica placement algorithm is to determine which nodes should host replicas. While choosing these nodes, the algorithm optimizes on a certain metric, such as client-to-replica latency. Deciding on replica placement, however, typically requires that other factors are considered as well, such as the amount of storage space available at candidate replica-holding nodes, or their network connection speed.

To solve this problem, we take a two-step approach. The first step is to select *network regions* where replicas should be placed. A network region is a group of nodes whose latencies to each other are relatively low. Since other node properties are irrelevant for region selection, only the relative latencies between nodes must be analyzed at this stage.

Once the network regions have been identified, the second step is to choose individual nodes that shall host replicas in different regions. During node selection, we consider various node-specific factors ignored during the region selection. However, because node selection takes place inside a region, the number of nodes that must be considered is much smaller, which makes the problem easier to solve. We return to the problem of intra-region node selection in Chapter 5. Note that this two-step approach prioritizes client-to-replica latency over any other metrics that depend on node-specific factors, which is in line with the overall goal of latency-driven replica placement.

This chapter discusses the region-selection algorithm, called HotZone. For evaluation purposes, we assume that the complementary algorithm for node selection within a region simply chooses the node with minimal average distance to all other nodes in the region.

4.3.1. Region selection

HotZone places replicas in network regions based on the relative latencies between nodes. Essentially, it works similar to the HotSpot algorithm: it first identifies network regions, then orders the regions according to the load they generate, and finally places replicas one-by-one in subsequent regions starting from the most active region. Placing a replica inside a heavily loaded network region seems to be attractive, as it should improve the access latency for a large number of requests.

Identifying network regions means determining groups of nodes whose latencies to each other are relatively low. In general, it would require analyzing all pair-wise latencies between nodes, which is likely to be computationally expensive in large-scale systems.

To reduce the computational overhead, we identify network regions based on node coordinates produced by GNP. Recall that GNP approximates the latency

between two nodes with the distance between their corresponding coordinates in an M-dimensional Euclidean space. The main observation we make at this point is that node coordinates are not uniformly distributed over the Euclidean space. Moreover, each cluster of node coordinates denotes a highly concentrated group of nodes whose latencies between each other are very low. It is therefore natural to identify network regions by determining clusters of node coordinates in the Euclidean space.

A question remains how to identify and measure coordinate clusters. To this end, we split the entire M-dimensional space into *cells* of identical size. A cell is an M-dimensional hypercube whose edge length is equal to some fixed value C. Each cell is uniquely defined by its *center point*. Because of the geometric properties of our space partition, the coordinates of each center point are

$$(C.k_1 + \tfrac{1}{2}C, .., C.k_M + \tfrac{1}{2}C)$$

for some values of k_i, where i ranges from 1 to M. We identify each cell by means of the integer-valued vector $(k_1, .., k_M)$ yielding the center point of that cell. The *density* of a cell is defined as the number of nodes whose coordinates fall within that cell. Note that this definition can easily be extended to support different node weights depending on the load generated by each node. To calculate the densities of all cells, the coordinates of each node $(x_1, .., x_M)$ are mapped to their corresponding cells $(k_1, .., k_M)$ according to the formula

$$k_i = \lfloor x_i / C \rfloor \quad i = 1, .., M$$

In this way, N nodes can be mapped into their corresponding cells in $O(N)$ time.

In a straightforward approach to the cluster identification problem, we could treat each cell as a potential cluster, and place replicas in the most dense cells only. However, a problem that arises at this point is that clusters may span multiple cells. This happens when cell boundaries divide a coordinate cluster into several parts with each part falling into a different cell [see Figure 4.1]. A split cluster is less likely to be given a replica, since each of its parts can be too little to outweigh smaller, but undivided clusters. This could lead to suboptimal performance of our algorithm.

To alleviate the problem of split clusters, we introduce the notion of a *zone*. Each zone is uniquely defined by a cell, and consists of that cell plus all the immediate neighboring cells. Each zone contains therefore 3^M cells in total. In other words, a zone is a group of adjacent cells, which together form a hypercube with edge length $3C$. Each zone shares its center point with the cell that defines that zone. We define the density of a given zone as the sum of the densities of all its member cells. Since zones do overlap, the density of a single cell contributes to the densities of several zones.

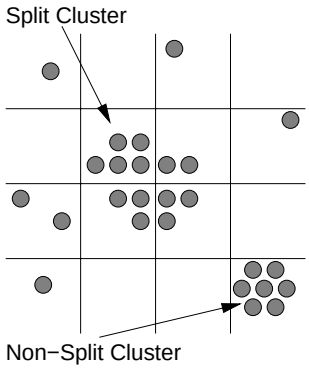


Figure 4.1: Split and non-split coordinate clusters in a 2-dimensional space

Operating on zones instead of cells does not completely solve the problem of split coordinate clusters, because coordinate clusters can still be large enough to be scattered over several disjoint zones. We return to this issue below, when discussing how to choose the cell edge length C.

When expressed in terms of zones, HotZone works similar to the Greedy algorithm. In every iteration, it identifies the most dense zone, marks this zone as a "replica holder," and removes all the node coordinates in this zone so that they are not considered in the remaining iterations. In this way, we implicitly assign the removed nodes to the replica that is currently being placed. More importantly, however, we reduce the possibility that replica-holding zones overlap, as empty cells are unlikely to fall within a zone considered to be dense. This ensures that all the replica-holding zones together cover as many node coordinates as possible, and that the replicas are ultimately placed in different parts of the system.

4.3.2. Cell size choice

The notion of zone allows us to reduce the problem of identifying coordinate clusters to that of identifying cells that yield dense zones. Note that there is no one-to-one correspondence between zones and coordinate clusters. Depending on the cell edge length C, a zone can contain several coordinate clusters, or a coordinate cluster can be scattered over multiple disjoint zones. Therefore, selecting the value of C plays a key role for the efficiency of HotZone.

There are two factors on which the cell edge length C should intuitively depend. The first factor that influences it is the number of replicas K. Recall that HotZone effectively splits the entire space into K parts, and assigns each part to a different replica. The larger the value of K, the smaller the parts should be that are produced by the algorithm. Given how HotZone works, each such part should

ideally be identified as a separate zone. The cell edge length C should therefore be inversely proportional to the number of replicas K, which is given to HotZone as a parameter.

The second factor that should affect the cell edge length C is the distribution of node coordinates in the space. Since we are using zones to identify coordinate clusters, it is natural that the cell size depends on the typical cluster size. For example, if most node coordinates fall in a small number of dense clusters, then these clusters can be identified with a small cell edge length C. On the other hand, if node coordinates are more evenly dispersed over the entire space, then the cell edge length C must be longer. This is because the number of nodes falling in a single zone must be large enough to ensure that all the zones can be unambiguously ranked according to their density. Only then can HotZone correctly determine the most dense zones where replicas should be placed.

Coordinate distribution can be represented in different manners. We decided to use the *average* distance between node coordinates, as it is easy to calculate. Note that calculating the average distance typically requires calculating the distances between the coordinates of each node pair. This would require $O(N^2)$ operations, which we strive to avoid. In our case, however, it is enough to compute a good estimate of the average distance, as the positioning procedure itself is imperfect.

To obtain an estimate of the average distance, we calculate the average distance between an incrementally growing number of node coordinates until the resulting value stabilizes. Determining the stabilization point is not trivial, as the computed value changes up until all the node coordinates have been considered. However, as it turns out, if we take node coordinates in a random order, then the incrementally computed values quickly converge to the actual average distance.

More formally, we iteratively compute the average distance D_E between an incrementally growing number of nodes E:

$$D_E = \frac{2}{E^2} \sum_{1 \le i \le E} \sum_{1 \le j < i} distance(P(i), P(j))$$

where $P(k)$ denotes the coordinates of the k-th node in a randomly ordered table. We determine the stabilization point by checking whether the value of D_E has stabilized over the last 100 iterations. To do so, every 10 iterations, we calculate the difference between the maximum and minimum value of D_E that has occurred within the last 100 iterations, and verify whether it is less than some threshold value ε:

$$max(D_{E-99}, .., D_E) - min(D_{E-99}, .., D_E) < \varepsilon$$

If the threshold is exceeded, then we increase E and proceed with the next 10 iterations. Otherwise, we treat D_E as our estimate of the average distance.

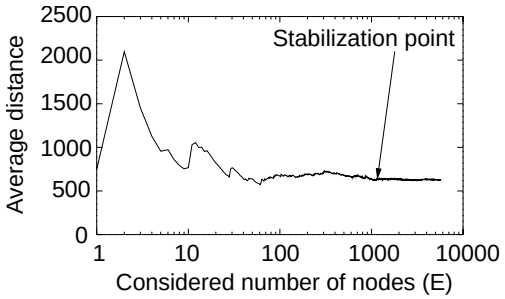


Figure 4.2: Incremental calculation of the average internode distance

We verified this method on a sample data set that contained 5728 node coordinates calculated in a 6-dimensional space. The positioned nodes were the Web clients that accessed our departmental Web server between the 1st and 30th of November 2003. The node coordinates were produced by a GNP instance we deployed. The instance was configured to use 12 PlanetLab nodes as landmarks [Bavier et al., 2004].

According to our experiments, the value of D_E converges after $E = 1110$ iterations when estimating the average distance between 5728 node coordinates with the ε value set to 10 [see Figure 4.2]. As we show in Section 4.4, similar numbers of iterations are sufficient to calculate the average internode distance also for significantly larger sets of node coordinates.

In order to discover how the number of replicas and the average internode distance contribute to the cell edge length C, we empirically determined the optimal values of C in a wide range of scenarios. We then used nonlinear regression to determine a function which outputs a good value of C for any combination of K and D. Considering that C is expected to be inversely proportional to the number of replicas K, and proportional to the average distance D, we decided to investigate the following family of functions:

$$C = \alpha \cdot \frac{D}{K^{\beta}}$$

where α and β are the coefficients that we need to determine.

To obtain the optimal values of C, we repetitively applied HotZone to the sample data set for each number of replicas $2 \leq K \leq 20$. We ignored the case when $K = 1$, as the best location is obviously the node whose average distance to all the other nodes is minimal. For each value of K, we repetitively ran HotZone with values of C ranging from 5 to D, taken in steps of 5. For each value of C, we evaluated the resulting placement by calculating the median distance between all nodes and their closest replicas. The C value yielding the shortest median distance was considered to be the best for the given number of replicas K. The outcome of

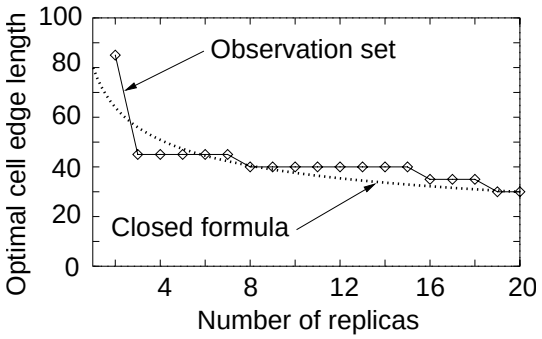


Figure 4.3: The (*K*,*C*) observation set and its approximation

this experiment was a set of observations, each being a pair of (*K*, best value of *C*). This set is depicted in Figure 4.3.

To obtain a closed function formula, we applied the nonlinear regression algorithm implemented in Gnuplot to our set of observations. The resulting values of α and β were 0.126 ($\approx \frac{1}{8}$) and 0.329 ($\approx \frac{1}{3}$), respectively, which gave us the following closed formula for the cell edge length *C*:

$$C \approx \frac{1}{8} \cdot \frac{D}{\sqrt[3]{K}}$$

where *D* is the average distance between nodes, and *K* is the number of replicas. The corresponding function is plotted in Figure 4.3. Note that since we computed the set of observations based on a single data set, we used the set-specific value of the average internode distance *D* equal to 670. As we show below, however, the resulting closed function formula works also for other data sets, for which the value of *D* is completely different.

Interpreting the values of α and β is difficult. As for the value of α, we believe that it is determined by the uneven distribution of nodes over the space. The value of β, however, required a more detailed investigation. Before running the experiments, we expected β to be $\frac{1}{6}$ (rather than $\frac{1}{3}$), which would correspond to the space dimensionality. Our intuition was that, if the replicas are distributed more-or-less evenly over the space, then they will probably themselves form a regular structure similar to a hypercube. For example, if we evenly divided a square (2D) space among 4 replicas, then the replicas would form a square. Similarly, if we were placing 8 replicas in a cubic (3D) space, then the replicas would form a cube [see Figure 4.4]. In both cases, the zone edge length would be the space edge length divided by $\sqrt[M]{K}$, where *K* is the number of replicas, and *M* is the space dimension.

The reason why this intuition is wrong is that spaces produced by our GNP implementations have dimensions of very different "width." By the width of the

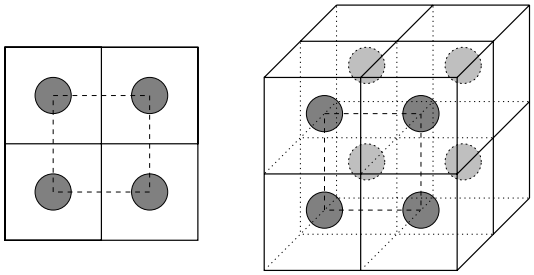


Figure 4.4: Theoretical even replica distribution

i-th dimension we mean the span of node coordinates along the *i*-th coordinate. Wide dimensions contribute to the actual distance between nodes, as they allow for large differences between node positions. Narrow dimensions, in turn, only slightly change node positions, but are important for the accuracy of latency estimation. However, because HotZone groups nodes that have similar node coordinates, it ignores narrow dimensions and benefits only from wide ones. As it turns out, the space from which we derived the observation set contained only three broad dimensions with widths of 4920, 3840, and 1855 milliseconds (and three narrow ones withs widths of 320, 310, and 10 milliseconds). This may indicate why the value of β is approximately $\frac{1}{3}$ instead of the expected $\frac{1}{6}$.

4.3.3. Complexity analysis

In this section, we analyze the computational complexity of HotZone. Similar to the notation used in the previous sections, we use N to denote the number of nodes, K to denote the number of replicas, and M to denote the GNP space dimension.

The computational cost of HotZone consists of three parts, each corresponding to a single step. The first step is to determine the average distance between nodes. As we show in Section 4.4, HotZone obtains a good estimate of this distance by calculating the distance between a fixed number of randomly selected nodes. The cost of this operation is constant.

The second step is to construct the set of zones. HotZone first assigns nodes to their corresponding cells, which costs $O(N)$ operations. Then, the set of non-empty cells is translated to the set of zones. To do that, HotZone identifies the neighboring cells of each cell, and sums their densities. Given that each zone contains a constant (3^M) number of cells, and that the number of cells cannot exceed N, this operation costs $O(N)$ cell-accesses. In our implementation, we sort all the cells according to their center points right after all the nodes have been assigned to their cells. We do so using Radix sort, which costs $O(N \cdot M) = O(N)$ operations. Then, we access individual cells using binary search, which yields the

cost of $O(logN)$ per a cell access. This results in the total cost of the second step being $O(N \cdot logN)$.

The third step is to iteratively place replicas. For each replica, we identify the most dense zone, which requires inspecting all the zones. Since the number of zones cannot exceed N, the identification of the most dense zone costs $O(N)$ operations in the worst case, as we process all the cells in our cell table directly, and without using binary search this time. Given that the same operations are performed for each replica, the total cost of the third step is $O(K \cdot N)$.

A potential source of overhead can be the update of zones. Recall that, in every iteration, once a replica has been placed in the most dense zone, we have to prevent the nodes in that zone from being considered in the subsequent algorithm iterations. This means updating not only all the 3^M cells that belong to the most dense zone, but also all the neighboring cells of these cells, as the density of zones yielded by the neighbors must be updated as well. This leads to the update of up to 3^{2M} cells. Although this constant is large, it makes HotZone independent of the number of nodes. This makes HotZone particularly efficient in large systems. In our implementation, each update costs $O(logN)$ operations, as we access the cell in question using binary search. Still, however, the cost of placing a single replica is $O(N)$.

The total computational cost of HotZone is the sum of the costs of the above three steps:

$$O(1) + O(N \cdot logN) + O(K \cdot N) = O(N \cdot max(logN, K))$$

This cost is significantly lower than that of the previously proposed algorithms, such as greedy ($O(K \cdot N^2)$) and HotSpot ($O(N^2 + min(N \cdot logN + N \cdot K)$).

4.4. EVALUATION

We evaluate HotZone based on three data sets produced by our GNP instance [Szymaniak et al., 2004]. It positions Web clients in a 6-dimensional space based on latencies between these clients and a number of Web servers acting as landmarks. Latencies are measured while Web clients open HTTP connections to the Web servers in order to retrieve small images. We embedded references to these images in the Web documents constituting the three Web sites participating in our node-positioning experiment. In this way, our GNP instance is able to position Web clients visiting three independent Web sites, which we treat as three different data sets [see Table 4.1]. All the measurements used to produce these data sets were performed between the 1st of February and the 31st of March, 2004.

Dataset	Web Site Description	Unique IPs	Client Profile
Andy	Andrew Tanenbaum's home page	5,758	Universities worldwide
Cartoon	Looney Tunes™ fan site	14,682	US schools
MP3	Dutch MP3 fan site	64,041	European DSL users

Table 4.1: Dataset statistics

Dataset	Nodes Total	Nodes Used	D_{Real}	D_{Calc}	Calc. Time
Andy	5,758	740 (12.85%)	602	580	92 msec
Cartoon	14,682	650 (4.42%)	385	393	71 msec
MP3	64,041	820 (1.28%)	290	290	113 msec

Table 4.2: Incremental calculation of the average internode distance

4.4.1. Average distance calculation

Determining the right cell size is crucial to good behavior of HotZone. Since the cell size depends on the average distance between nodes, we first investigated how the incremental calculation method performs on our data sets. The results are presented in Figure 4.5 and Table 4.2.

As can be observed, irrespective of the size of the data set, a similar number of nodes is necessary to calculate a good estimate of the average distance. This observation confirms that this cost should be treated as a constant in the evaluation of the computational complexity of HotZone. Furthermore, this constant time is very small compared to the overall time of replica placement computation (about 100 milliseconds vs. several seconds).

4.4.2. Closed formula verification

HotZone calculates the cell size using a formula that depends on two parameters, α and β. Recall that in Section 4.3, we determined α and β by applying nonlinear regression to a set of optimal cell sizes, which we computed based on a single sample data set. In this experiment, we verify how general these α and β values are by computing them for each of the other three data sets. The results are presented in Table 4.3.

As can be observed, the values of α and β computed for the Andy and MP3 data sets are very similar to their counterparts derived from the sample data set. However, it is clearly not so for the Cartoon data set, where the values of α and β are significantly different. We believe that this is because of the fact that nodes in the Cartoon data set are more evenly distributed over the space. In that case, clients barely form any clusters, which leads to an optimal cell size significantly

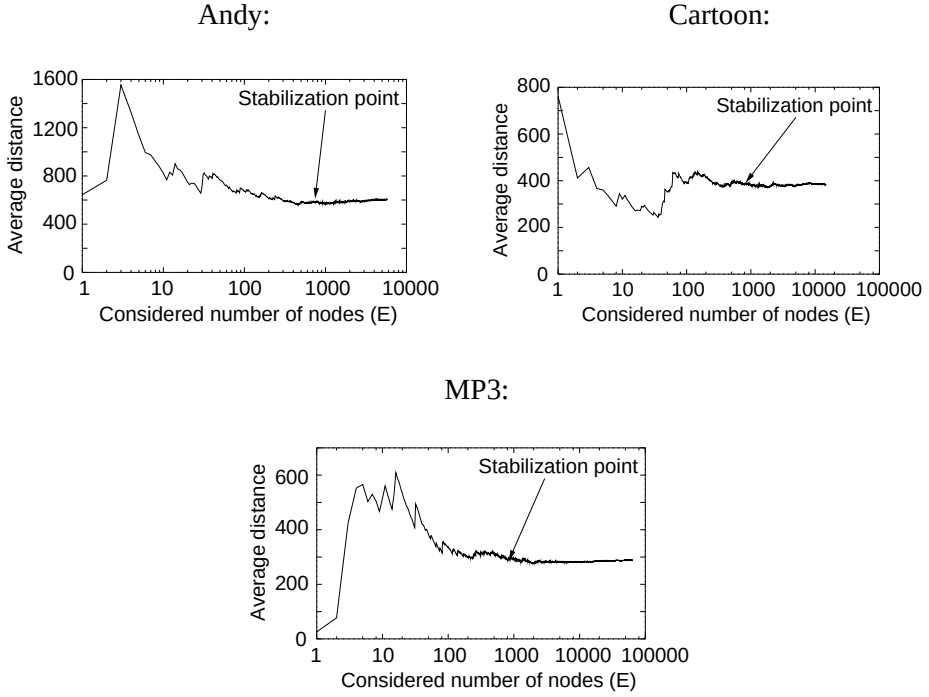


Figure 4.5: Incremental calculation of the average internode distance

larger than expected. However, as we show in Section 4.4.3, even sub-optimal α and β parameters lead to placements almost as good as those produced by Greedy.

According to our calculations, using the data set-specific parameters to compute placements for the Cartoon data set would improve the placement quality by at most 10%. This could indicate that the quality of placements is quite resilient to changes in the values of α and β, or even in the value of C itself.

To verify this claim, we checked by how much imperfect values of C influence the quality of placements. We again analyzed the results of the experiment where we repetitively placed replicas using all possible C values between 5 and 100. Previously, we only used the C value yielding the minimal median latency. This time, however, we also checked how fast this median latency increases as the C value drifts away from the optimal one. Figure 4.6 shows median latencies computed for different C values. For sake of clarity, we only show the results for placing 5, 10, and 20 replicas based on the Andy data set. Other data sets exhibit similar behavior.

As can be observed, the optimal C value belongs to a relatively long interval, where each value yields a median latency within 10% of the minimal one. According to our experiments, the C values calculated by HotZone belong to that interval.

Dataset	α value	β value
Sample	0.126	0.329
Andy	0.154	0.310
Cartoon	0.363	0.651
MP3	0.130	0.373

Table 4.3: The values of the α and β coefficients

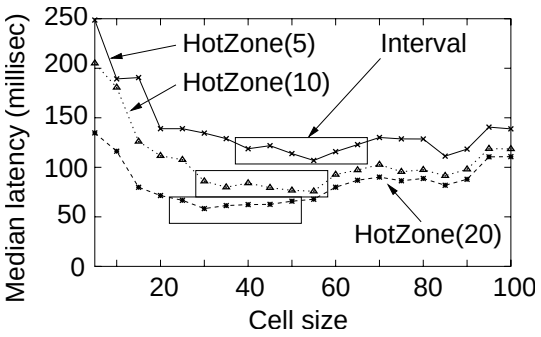


Figure 4.6: Changes in placement quality vs. *C* values

This explains why HotZone produces near-optimal placements even though the *C* values themselves may be imperfect.

4.4.3. Placement quality comparison

A popular method of evaluating replica placements is calculation of the *average* latency between nodes and their closest replicas. However, the results produced by this method very much depend on outliers, which are nodes whose latencies to any other nodes are very high. Outliers typically use slow network connections, such as modems, and do not tend to form clusters. Because of these two factors, it is hard to improve the replica-access latencies of outliers without placing replicas specifically on them. We therefore decided to concentrate on the remaining nodes by evaluating placements with the *median* latency between nodes and their closest replicas.

We evaluated HotZone against the Greedy and HotSpot algorithms described in Section 4.2. Recall that HotSpot needs to know how to determine whether two given nodes are neighbors or not. We configured HotSpot to consider nodes to be neighbors only if the distance between them is less than some threshold value. To ensure the fairness of comparison, we tried various threshold values and report only the best result.

We also considered a variant of HotSpot, which we call "HotClear." After placing a replica on a given node, HotClear removes all the nodes from the neighborhood of this node so that they are not considered in the subsequent iterations. Also in this case, we tried several threshold values and report only the best result.

Ideally, we should also compare HotSpot against placements yielding optimal median node-to-replica distance. However, since computing optimal placements is unfeasible, we decided to approximate them using Simplex-downhill, which is a multi-dimensional optimization algorithm [Nelder and Mead, 1965]. When applied to the placements produced by HotZone, Simplex-downhill investigates similar replica locations, and tries to move replicas so that the median node-to-replica latency is reduced. We refer to this method as "Refined." Simplex-downhill is computationally expensive, so it is unlikely that Refined can be used in practical applications. Nevertheless, the comparison between HotZone and Refined enables us to estimate by how much the original placement can be improved.

We apply the five evaluated algorithms to each of the three data sets. We iteratively place K replicas, for K between 1 and 20, and calculate the median node-to-replica latency for each value of K. The results are presented in Figure 4.7.

As can be observed, the original HotSpot algorithm performs significantly worse than all the others. This is not surprising: as it turns out, HotSpot places replicas on nodes whose coordinates are close to the centers of a few very active neighborhoods. Although these neighborhoods may change depending on the threshold value, the replicas are ultimately placed close to each other, which results in poor performance. This is exactly the effect that HotZone tries to avoid by using overlapping network regions.

The remaining four algorithms produce comparable results. Compared with Greedy, HotZone offers median latency that remains within 13%, 14% and 9% of that offered by Greedy for the Andy, Cartoon, and MP3 data set, respectively. The average difference between latencies offered by these two algorithms, however, is between 3% and 5% in all the data sets. Note that HotZone sometimes slightly outperforms Greedy, which can be seen in the graph depicting the replica placement based on the MP3 data set.

HotZone usually performs better than HotClear. In this case, however, the average difference in latencies is very small (between 5% and 7%) for the two smaller data sets, but it increases to 16% for the biggest data set. The case of HotClear shows that the original HotSpot algorithm can easily be improved to work effectively on node coordinate sets, but even then it still cannot outperform HotZone.

In comparison with Refined, HotZone obviously performs worse. However, the difference in the offered median latency is not large and on average equals 8%, 12% and 5% (in the respective data sets). This may indicate that the placements

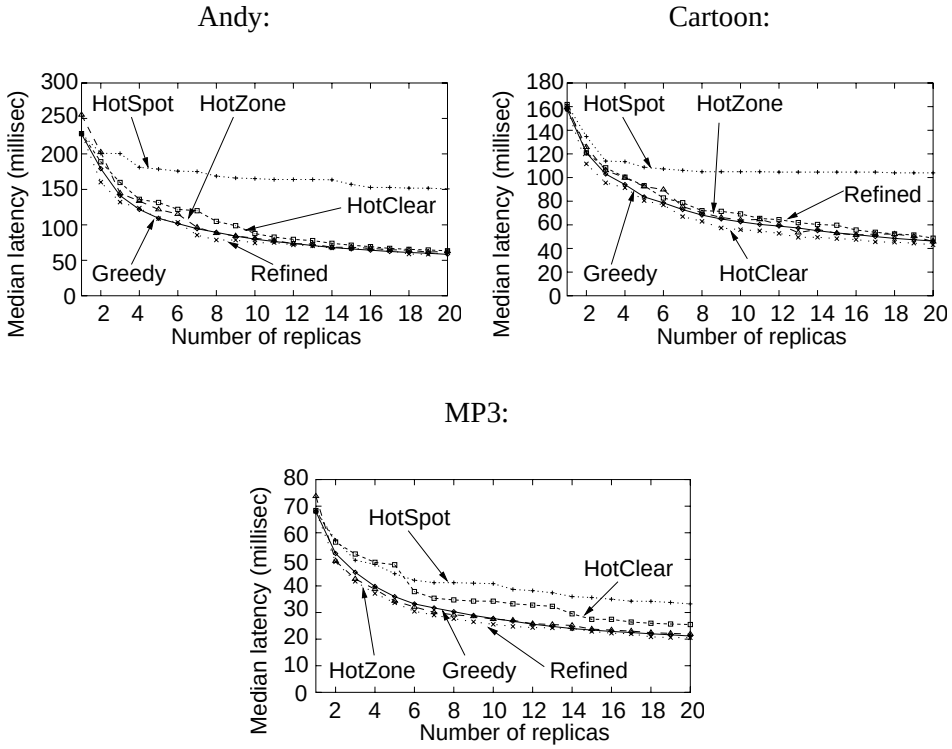


Figure 4.7: Placement quality comparison

produced by HotZone cannot be improved much, especially that they are already comparable in terms of quality to those produced by Greedy.

4.4.4. Placement computation times

The main advantage of HotZone over other algorithms is its low computational cost. In this experiment, we show how the differences in computational complexities of different algorithms translate into placement computation times. We measured the execution times of the five evaluated algorithms for each number of replicas K between 1 and 20, based on all the three data sets. The tests were performed on an idle PC equipped with a Pentium III 1GHz. The results are depicted in Figure 4.8 (note the logarithmic time scale).

As can be observed, the time needed by HotZone to compute its placements up to 3 orders of magnitude lower than the time needed by Greedy. As for Refined, the unpredictable nature of Simplex-downhill resulted in irregular execution times. Still, it is interesting to see that Refined is consistently faster than Greedy, even though Refined nearly always produces better results than Greedy.

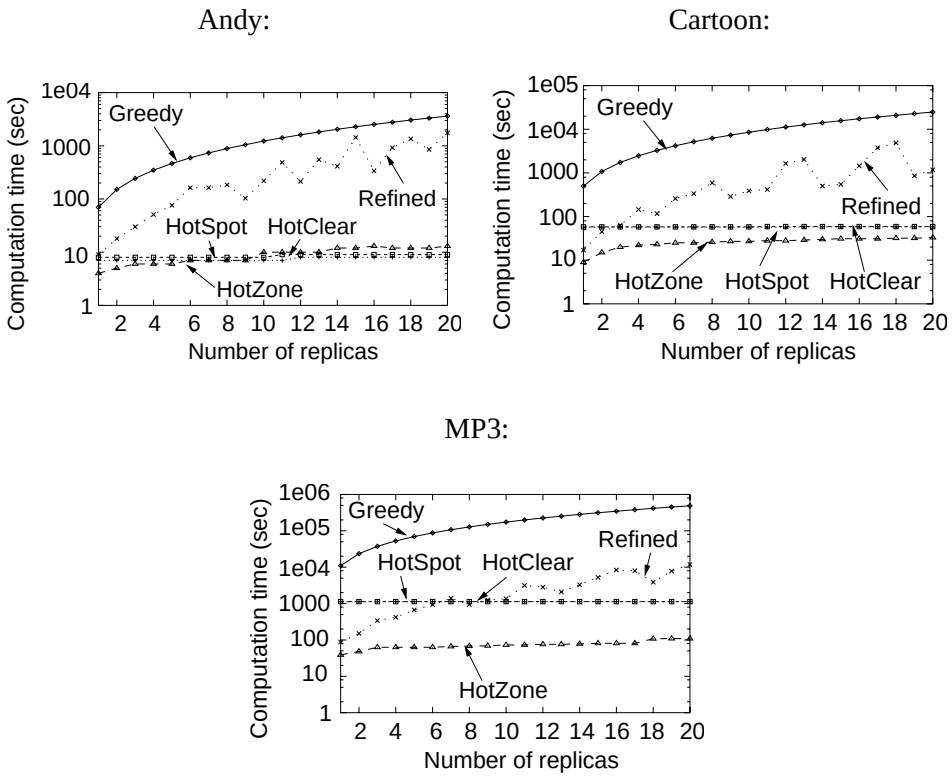


Figure 4.8: Placement computation times

In comparison with HotSpot and HotClear, the computation time of HotZone is better only for the two larger data sets, and the advantage of HotZone over these two algorithms increases with the number of nodes in a data set. For the smallest data set, however, the computation time of HotZone is comparable to these of HotSpot and HotClear. This indicates that HotZone is particularly suitable for large-scale systems, where the number of nodes is very high.

4.5. CONCLUSION

We have presented HotZone, a novel replica placement algorithm that optimizes node-to-replica latency based on node coordinates produced by GNP. Similar to the previously proposed HotSpot algorithm, HotZone places replicas on nodes that along with their neighboring nodes generate the highest load. In contrast to HotSpot, however, HotZone does not require that $O(N^2)$ operations are performed to determine the neighborhood composition for all the N nodes in a

given system. Instead, it exploits the properties of node coordinates to determine the composition of all neighborhoods faster. We have demonstrated that the computational cost of placing K replicas using HotZone is $O(N \cdot max(logN, K))$, which is significantly lower than that of any previously proposed algorithm. This makes HotZone attractive for use in large-scale distributed systems.

HotZone produces results of comparable quality to those of Greedy. Furthermore, for large data sets, the computation time of HotZone is significantly lower than those of the other evaluated algorithms. In particular, it is up to 3 orders of magnitude lower than the computation time of Greedy.

HotZone implements the first step of our two-step solution for global replica placement by identifying the network regions where replicas should be placed. The second step is to select individual nodes within these regions to host actual replicas. However, as large-scale distributed systems increasingly often consist of low-performance nodes, it might be difficult to find suitable candidates for replica hosting. In such cases, it is desired to organize all candidate nodes in a single region into a distributed server, which autonomously manages its hosted replicas. In particular, the distributed server can decide on which node each replica should be hosted depending on individual characteristics of each node, such as availability and bandwidth. It might also decide to host multiple internal replicas to preserve their content despite failures of individual nodes. The next chapter discusses a number of techniques that enable turning multiple nodes scattered over a wide-area network into a single distributed server. We demonstrate how to equip each such distributed server with a single contact address, and how to transparently switch clients between individual nodes forming that distributed server.

CHAPTER 5

Resource Aggregation

5.1. INTRODUCTION

With the growing popularity of Internet services, many services need to adapt their operation in order to preserve short response times. Instead of running on a single node, such services often move to distributed infrastructures in which requests originally serviced by a single server are processed by many nodes, each handling only a subset of requests. Distributing the request load over multiple nodes generally increases the processing capacity of an Internet service.

Apart from increasing their capacity, modern Internet services also strive to reduce the communication delays experienced by their clients. As we explained in Chapter 1, communication delays depend to a large extent on network latencies between an Internet service and its clients, as such latencies determine both the service access delay and the maximum throughput of client TCP connections. Minimizing the network latencies accelerates the communication with the service, thus improving client-perceived performance.

This thesis proposes to optimize network latencies by means of replication, in which multiple service replicas are deployed in various parts of the Internet. Each replica can then handle requests sent by clients in its proximity, which shortens the network paths traversed by client requests, effectively reducing the network latencies. Such latency-driven replication requires that replica locations are carefully selected based on estimated latencies between clients and replicas.

The previous chapter proposed the HotZone algorithm, which identifies good locations ("network regions") for replicas based on latency models produced by GNP. However, as HotZone focuses on latency-driven replica placement, it ignores node properties other than their latencies to other nodes. Meanwhile, once the issue of minimizing client-replica latencies has been solved, other replica properties turn out to be important as well. For example, replicas should be highly

available in order to provide uninterrupted service to their clients. Furthermore, replicas should also have fast network connections in order to fully exploit the potential of high TCP connection throughput resulting from short latencies. These properties should be taken into account when selecting the replica-hosting nodes within individual network regions.

However, in many network regions, locating highly available nodes with fast network connections might turn out to be infeasible. This is particularly true when running an Internet service on relatively weak machines contributed by volunteers, in which case neither high availability nor fast network connections can ever be guaranteed. A good example of such a system is a peer-to-peer content delivery network such as Globule [Pierre and van Steen, 2006].

We propose to compensate for the lack of powerful replica hosting facilities in a network region by enabling an Internet service to *construct* such facilities out of whatever nodes are available in that region. To this end, we propose to organize all these nodes into a distributed replica hosting infrastructure that can meet the service requirements with respect to availability and bandwidth by aggregating the resources provided by individual nodes.

The problem with hosting a service replica on a distributed infrastructure is that using it must not result in breaking the classical service access model adopted by client-side software. In other words, the distributed infrastructure still needs to provide clients with a single contact address to which requests can be sent. Traditionally, this is achieved by appointing one of the nodes within the infrastructure as a frontend, which advertises its physical address to the clients, receives all the client requests, and dispatches them to other nodes [Cardellini et al., 2002].

Employing a frontend results in hiding the distributed nature of a replica hosting infrastructure from the clients and separates the infrastructure design from the single-address access model adopted by the client-side software. On the other hand, as frontends proxy all the client communication, they can also become potential performance bottlenecks [Brewer, 2001]. This is particularly likely to happen in our case, as we assume the lack of sufficiently powerful nodes in the network region. Deploying a frontend on a slow or unreliable node, in turn, would most likely result in poor service performance, and could even lead to a complete replica outage when the frontend goes down.

We propose to enable the deployment of Internet services on collections of weak or unreliable nodes by eliminating the need for a frontend while preserving the transparency of infrastructure distribution to the client-side software. The idea is to identify an Internet service with a single stable *logical* network address. Similar to traditional service addresses, a logical address can always be used to communicate with the service. The difference, however, is that logical addresses are not bound to any physical node. Instead, each such address can be dynamically

mapped to any node within the service infrastructure while the service retains full and timely control of this mapping.

Assigning a logical address to a group of physical nodes comes close to anycasting. Anycast was proposed as a routing and addressing scheme by which traffic sent to an anycast address automatically reaches some node within the addressed group [Partridge et al., 1993]. This functionality is typically implemented by means of routing algorithms, which cause Internet routers to redirect anycast traffic according to some network proximity metric.

However, efficient usage of resources available on nodes located within the same network region depends on many factors other than network proximity. For example, differences in node utilization make it necessary to route client requests according to additional metrics such as network bandwidth and CPU load. Also, as the utilization and availability of nodes change dynamically, the routing decisions must have immediate effects to prevent client requests from being redirected to overloaded or unavailable nodes [Sarat et al., 2005]. Finally, anycasting should not introduce significant overhead compared to unicast communication.

The limitations of routing-based anycast has led to proposing many alternative implementations of anycast-like functionality, including the use of either front ends [Cardellini et al., 2002], DHTs [Castro et al., 2003], DNS-based redirection [Zegura et al., 2000], or anycast-aware client-side software [Fei et al., 1998]. However, as we discuss in a recent study, none of these proposals are actually satisfactory enough [Szymaniak et al., 2006b].

Our solution lies in the design of *versatile anycast*, in which each anycast group retains full and timely control over the way the incoming traffic is switched among the individual nodes within that group. At the same time, our implementation does not incur any significant communication overhead compared to unicast communication. These two properties enable us to implement the logical addresses of Internet services as anycast addresses provided by versatile anycast.

We implement versatile anycast by exploiting the logical separation of network addresses that Mobile IPv6 assigns to mobile nodes. In principle, each Mobile IPv6 node has a permanent "home" address, which identifies the node, and a temporary "care-of" address, which identifies the node's current location. Mobile IPv6 ensures that network traffic sent to home addresses is transparently forwarded to their care-of counterparts. To this end, it relies on clients communicating with mobile nodes to translate between home and care-of addresses.

This chapter demonstrates that the very same translation mechanisms can also be used to equip (the replicas of) Internet services with logical addresses. In that case, an Internet service as a whole can be perceived by its clients as a single fictitious mobile node, regardless of the current composition of the service infrastructure. The logical address of the service is implemented as the home address

of the fictitious mobile node, whereas the addresses of individual nodes within the service infrastructure can be treated as potential care-of addresses of that node. Using traffic-switching mechanisms provided by Mobile IPv6, an Internet service can transparently handoff its clients to individual nodes at which it is hosted.

Our approach has several advantages. First, implementing logical addresses in the network layer allows for leaving the higher layers of client-side software untouched. This means in particular that with a relatively small number of server-side modifications, our scheme can be incorporated into *any* service exploiting the traditional access model based on single contact addresses, including those that already exist. Furthermore, dynamic traffic switching enables each node to handoff its clients to other nodes within the service infrastructure at any moment. This feature allows for deploying Internet services on dynamic collections of nodes, as each such node can seamlessly leave the infrastructure provided that all the clients handled by that node are handed off to other nodes in advance. Finally, as traffic switching is performed on a per-client basis, the same logical address can be shared by multiple nodes communicating with different clients. This effectively leads to aggregating the resources contributed by individual nodes (e.g., bandwidth) and making them all available at a single logical address, just as if they were provided by a single powerful node. We believe that such an ability to resource aggregation is an important step towards constructing reliable and high-performance hosting facilities for Internet services.

The possibility of implementing anycast functionality using Mobile IPv6 has been identified in two earlier publications. The first one proposes to exploit mobile extensions of IPv6 to route requests in content delivery networks [Acharya and Shaikh, 2002], whereas the second one sketches how to redirect clients to anycast nodes using Mobile IPv6 signaling [Haberman and Nordmark, 2002]. However, both these studies build on early versions of the Mobile IPv6 specification, which differ significantly from the final protocol analyzed in our research. Also, besides employing Mobile IPv6 to implement (relatively straightforward) one-time traffic switching, we also demonstrate how Mobile IPv6 can be exploited to ensure anycast address stability and to implement multi-layer wide-area client handoffs. Finally, while the two earlier studies are purely theoretical in nature, we base our considerations on practical experience with our prototype anycast testbed.

The remainder of this chapter is structured as follows. Section 5.2 describes the architecture of Internet services equipped with logical addresses. Section 5.3 presents related work and explains why it is hard to provide logical addresses using current techniques. Section 5.4 describes how Internet services can exploit versatile anycast to implement logical addresses, and how versatile anycast can be implemented using Mobile IPv6. Finally, Section 5.5 evaluates the performance of our anycast implementation, and Section 5.7 concludes.

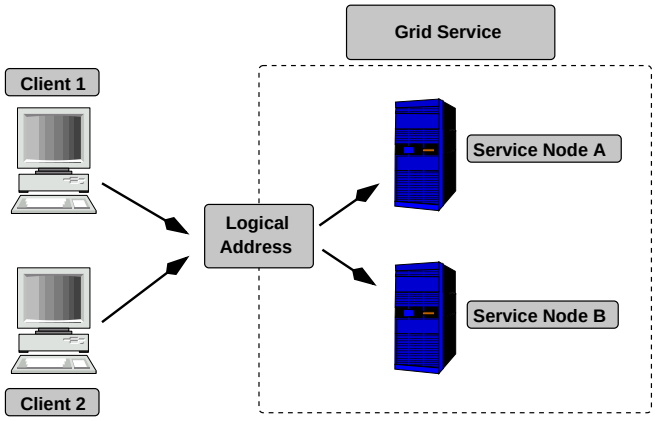


Figure 5.1: Accessing Internet services via logical addresses

5.2. SYSTEM MODEL

5.2.1. Overview

Introducing logical addresses enables Internet services to decouple the client-side software development from the service-side infrastructure design. Figure 5.1 depicts the conceptual service access model. In principle, nothing changes from the perspective of the service clients, which access the service at its logical address just like they traditionally communicate with the addresses of frontends.

However, the logical address is not bound to any physical node. It is therefore the responsibility of the service to ensure that all the traffic heading to that address is routed to the physical address of one or more nodes within the service infrastructure. To this end, the service transparently maps the logical address onto the physical addresses of the service nodes. As long as the mapping mechanism enables the service to dynamically switch the clients among the service nodes, each of these nodes can join and leave the service infrastructure at will, which increasing the infrastructure adaptability.

5.2.2. Properties

The functionality of logical addresses requires them to have a number of properties. The first fundamental one is transparency, which means that using logical addresses cannot mandate any changes to the client-side software, which must be able to communicate with Internet services via logical addresses just like via their physical counterparts. The second fundamental property is efficiency, which means that accessing Internet services via logical addresses should not incur sig-

nificant overhead compared to doing so via physical addresses. In particular, the clients should be able to communicate efficiently with an Internet service even when service nodes are scattered over a wide-area network, which is often the case with massively popular network services [Barroso et al., 2003].

Another group of properties is related to the mapping of logical addresses onto physical nodes. For example, efficient usage of service nodes requires fine-grain control over which clients are redirected to which nodes. This means that the logical address implementation must enable the service to redirect each client separately and according to any set of metrics. For example, classical load-balancing schemes route traffic based on the current load of each service node, and on the network distance between clients and service nodes [Rabinovich and Aggarwal, 1999; Cardellini et al., 2003].

Another characteristic of modern Internet services is that they are running on large collections of nodes that can dynamically join and leave the service infrastructure [Foster and Iamnitchi, 2003]. As a consequence, the service infrastructure might experience frequent changes in its hardware composition. However, such changes should not affect the service performance, and so the service should be able to quickly adapt to sudden departures of service nodes. This can be achieved by transparently redirecting clients from the departing nodes to those remaining operational, which requires that each client can be switched from one service node to another at any moment.

However, while switching traffic is relatively easy when clients communicate with Internet services using connectionless protocols, it becomes far more complex when connection-oriented protocols, such as TCP, are used for this purpose. This is because these protocols require some state information to be maintained by both the clients and the service nodes. To guarantee that client connections are not broken upon switching, the logical address implementation must ensure that traffic switching is coordinated with its corresponding state transfer between service nodes.

5.3. ALTERNATIVE SOLUTIONS

There exist several techniques that enable service adaptability by means of logical addresses. A number of them achieves that by mandating modifications to the client-side software such as those organizing clients and service nodes into P2P overlays [Ganguly et al., 2006; Chun et al., 2006; Castro et al., 2003]. However, as one of our main goals is to keep the client-side software untouched, none of these solutions is attractive in our case.

Several techniques implement logical addresses without modifying the client-side software. The classical one implements a logical address as that of a physical front end, which forwards client traffic to individual nodes hosting an Internet service [Cardellini et al., 2002]. Such a solution offers real-time and fine-grain control over the client traffic. However, when used in wide-area setups, front ends tend to become performance bottlenecks, as they limit network bandwidth available to the service and introduce additional client access latency [Brewer, 2001].

Another common solution is to map clients to service nodes using DNS. In that case, each service is identified by its DNS name rather than a network address. The mapping of the service DNS name onto the addresses of service nodes is performed by the DNS server responsible for that name. In essence, this DNS server can resolve the service DNS name to the addresses of different service nodes such that the client requests are ultimately scattered over multiple service nodes [Brisco, 1995].

DNS redirection has been successfully employed by many large-scale distributed systems, as it integrates transparently into the Internet communication model, exploits the scalability of DNS, and provides fairly good control over client redirection [Dilley et al., 2002]. However, DNS caching can severely delay updating the redirection mappings, as many DNS servers are configured to ignore short TTL values. This makes DNS unattractive to adaptable Internet services, which need to tolerate rapid changes in their hardware configuration. Also, since DNS names are resolved only before the actual communication is initiated, they cannot be used to switch clients between service nodes while communication is already in progress.

Logical addresses could also be implemented by means of anycasting. Anycast is a network addressing and routing scheme whereby data sent to an anycast address are routed to one of many nodes within its corresponding anycast group [Partridge et al., 1993]. The chosen node is typically the “nearest” or “best” to the data sender as viewed by the network topology. The classical anycast implementation relies on routing algorithms, which cause Internet routers to redirect anycast traffic according to some network proximity metric.

Internet services could implement logical addresses as anycast addresses. In that case, all the nodes hosting a given service would form an anycast group, and the anycast implementation would naturally spread the client traffic heading to the logical address among these nodes. Anycasting would therefore implement the conceptual service access model described in Section 5.2.

Using anycast to implement logical addresses would preserve the traditional service access model based on a single contact addresses, as each client would communicate with an Internet service via the single anycast address of that ser-

vice. At the same time, the adaptability of the service infrastructure would be preserved as well, as anycast groups are by nature supposed to change dynamically. However, our earlier study demonstrated that none of the current anycast implementations can provide all the properties required of logical addresses, such as efficiency, timely and fine-grain traffic control, or connection-aware client switching [Szymaniak et al., 2006b].

The anycast-based approach has recently been followed in OASIS, which essentially provides a general-purpose anycasting functionality to Internet services [Freedman et al., 2006]. OASIS integrates multi-node services into a global infrastructure in order to perform accurate network measurements, which in turn allows for mapping service clients to their proximal service nodes. The strength of OASIS lies in its advanced mapping policy. However, as OASIS relies on standard redirection mechanisms such as DNS, it also inherits their limitations discussed above. In fact, our anycast implementation proposed in this paper could be used by OASIS as yet another redirection mechanism.

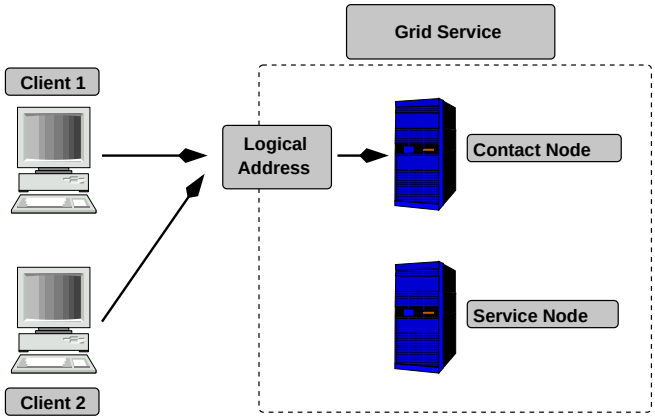
The following sections discuss in detail how to implement *versatile anycast*. In contrast to the previous anycast implementations, versatile anycast provides all the properties required of logical service addresses. We first discuss the architecture details, and then demonstrate that the overhead of versatile anycasting is very low.

5.4. VERSATILE ANYCAST

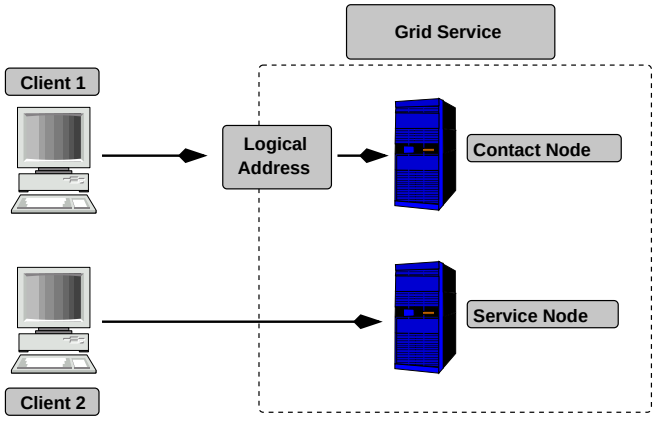
Versatile anycast allows Internet services to implement the conceptual service access model discussed in Section 5.2, in which the client traffic is redirected from the logical address of an Internet service to the physical address of some service node. Versatile anycast achieves that by implementing the logical address as an anycast address, and by switching the traffic heading to that address among the service nodes forming its corresponding anycast group.

Versatile anycast works in two phases. First, it ensures that the client traffic sent to the anycast address reaches a designated service node called a *contact node* [see Figure 5.2(a)]. This is achieved by assigning the anycast address to the contact node. However, to preserve service reachability even after the contact node becomes unavailable, versatile anycast allows the anycast address to be reassigned to any other service node at any moment, which effectively turns that node into a new contact node.

Of course, the contact node should not service all the clients by itself. Rather, it should distribute the client-handling effort among other service nodes. This constitutes the second phase of versatile anycasting, in which the contact node



(a)



(b)

Figure 5.2: Versatile anycast: establishing contact (a), and client handoff (b)

transparently hands off individual clients to other service nodes, potentially causing different clients to be serviced by different service nodes [see Figure 5.2(b)]. Note that once a client is handed off to some service node, the contact node is no longer involved in the communication with that client. Also, each service node can further handoff its clients to any other service node at any moment. These two features are crucial for service adaptability, as they enable each service node to share its load with new service nodes when they join the service infrastructure, and to leave the service infrastructure without disturbing its clients by handing them off before leaving.

We propose to implement versatile anycast using the address-translation capabilities provided by the Mobile IPv6 protocol. These capabilities have originally been introduced to enable communication with mobile nodes while they move among various networks. However, we demonstrate that one can also exploit these capabilities to implement versatile anycasting.

The following section discusses some basic aspects of Mobile IPv6, which is the standard protocol designed for mobile communication. Then, we show how selected functions of Mobile IPv6 can be employed to implement versatile anycast.

5.4.1. Mobile IPv6

Mobile IPv6 (MIPv6) consists of a set of extensions to the IPv6 protocol [Johnson et al., 2004]. MIPv6 has been proposed to enable any *IPv6 mobile node* (MN) to be reached by any other *correspondent node* (CN), even if the MN is temporarily away from its usual location.

MIPv6 assumes that each MN belongs to one home network, which contains at least one MIPv6-enabled router capable of serving as a *home agent* (HA). Such an HA acts as a representative for the MN while it is away.

An HA must authenticate MNs before it can start representing them [Arkko et al., 2004]. To this end, each MN must establish an *IPsec security association* with its HA in its home network [see Figure 5.3]. Such associations are established using the Internet Key Exchange [Harkins and Carrel, 1998].

To allow one to reach an MN while it is away from home and connected to some visited network, MIPv6 distinguishes between two types of addresses that are assigned to MNs. The *home address* identifies an MN in its home network and never changes. An MN can always be reached at its home address. An MN can also have a *care-of address*, which is obtained from a visited network when the MN moves to that network. The care-of address represents the current physical network attachment of the MN and can change as the MN moves among various networks. The MN reports all its care-of addresses to its HA.

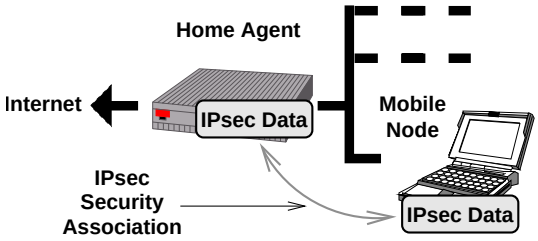


Figure 5.3: Home network in Mobile IPv6

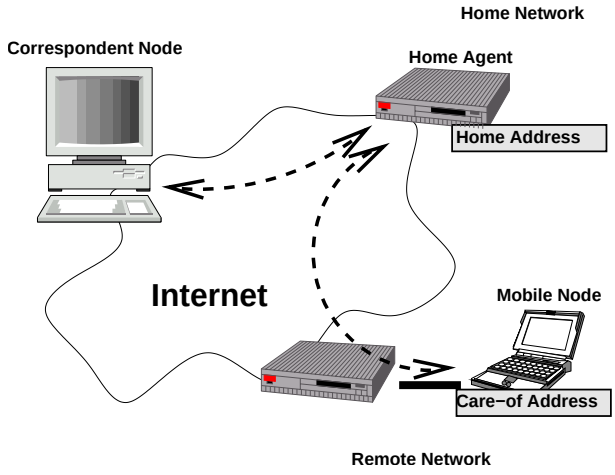
The goal of MIPv6 is to ensure uninterrupted communication with MNs via their home addresses and independently of their current network attachment. To this end, MIPv6 provides two mechanisms to communicate with MNs that are away from home. The first mechanism is *tunneling*, wherein the HA transparently tunnels the traffic targeting the home address of an MN to the care-of address of that node [see Figure 5.4(a)].

The advantage of tunneling is that it is totally transparent to the CNs. Hence, no MIPv6 support is required from any node other than the MN and its HA. However, tunneling can also lead to two problems. First, if many MNs from the same home network are away, then their shared HA can become a bottleneck. Also, if the distance between an MN and its home network is large, then tunneling can introduce significant communication latency. This is why we strive to avoid MIPv6 tunneling as much as possible, and instead focus on the second MIPv6 communication mechanism, called *route optimization*. It solves the two above problems by enabling an MN to reveal its care-of address to any CN, thus allowing direct communication between the MN and the CN [see Figure 5.4(b)].

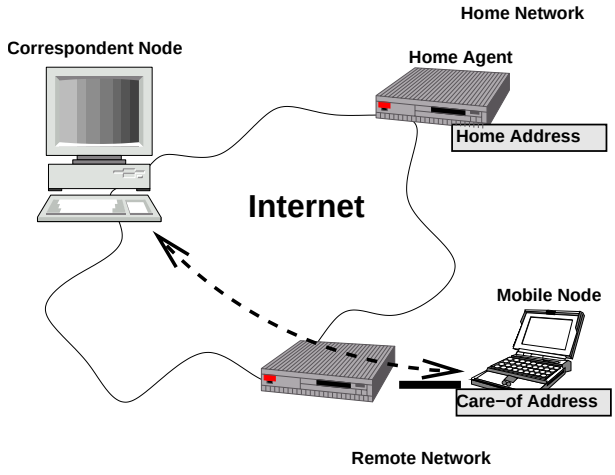
Route optimization is prone to address spoofing. To protect itself, the CN must authenticate the care-of address using a *return-routability procedure*, which is used to verify that the same MN can be reached at the HA and at the care-of address.

The return-routability procedure is initiated by the MN which simultaneously sends two messages to the CN [see Figure 5.5]. The first message, called *Home Test Init* (HoTI), is tunneled through the HA, whereas the second message, called *Care-of Test Init* (CoTI), is sent directly. The CN retrieves the MN's home address and care-of address from the first- and second message, respectively. The CN responds with two messages, *Home Test* (HoT) and *Care-of Test* (CoT). The HoT message is tunneled to the MN through the HA, whereas the CoT message is sent directly.

The HoT and CoT messages contain home and care-of keygen tokens, respectively, which are combined to create a *binding management key* (Kbm). The ability of the MN to create the Kbm based on the tokens received via two different paths



(a)



(b)

Figure 5.4: Communication in MIPv6: tunneling (a), and route optimization (b)

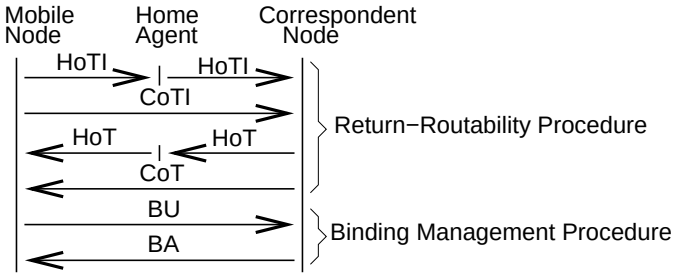


Figure 5.5: Route optimization protocol

is the proof that the MN has passed the return-routability procedure and that the home and care-of addresses correspond to the same MN.

The MN uses the Kbm to authorize the *binding management procedure*. The goal of this procedure is to create the mapping between home and care-of address at the CN such that it communicates directly with the MN. To this end, the MN sends the Kbm to the CN in a message called *Binding Update (BU)*. This message also contains the home address, the care-of address, the lifetime of the requested home-to-care-of address mapping, and a sequence number, which orders all the BU messages sent by a given MN to a given CN.

Upon receiving the BU message, the CN verifies that the Kbm found inside that message is valid and matches the home/care-of address pair. In this way, the CN can now be certain that the MN has passed the return-routability procedure. It therefore creates a *binding cache entry* for the MN, which is essentially a mapping between home and care-of address. The binding cache entry allows the CN to translate between home and care-of address in the incoming and outgoing traffic, which enables the CN to communicate with the MN directly at its care-of address. This eliminates the latency introduced by tunneling, and offloads the HA.

As the last step of route optimization, the CN confirms creating the binding cache entry by sending a *Binding Acknowledgment* (BA) message to the MN. Note that binding cache entries are deleted once their lifetime expires, and must be therefore periodically refreshed. The MN can also cause an old binding cache entry to be deleted immediately by sending a new BU message with the lifetime set to zero. Such a message can be sent without performing the return-routability procedure.

Route optimization is less transparent than tunneling, as the IP layer at the CN is aware of the current physical attachment of the MN. However, that information is confined inside the IP layer. The CN uses it to translate source and destination addresses in IP packets exchanged with MNs according to the binding cache entries created during the binding management procedures.

Translating addresses in the IP layer hides care-of addresses from higher-level protocols such as TCP and UDP. As a consequence, these protocols use only the home address of an MN and the changes in the MN's location remain transparent to applications running on CNs.

5.4.2. Anycasting with Mobile IPv6

Our implementation of versatile anycast exploits the fact that Mobile IPv6 decouples home and care-of addresses, effectively allowing for the traffic directed to the former to be transparently redirected to the latter. This comes close to the anycast communication model, in which traffic sent to the anycast address of an anycast group is routed to the interface of some node within that group. We exploit our implementation of versatile anycast to transparently redirect the clients of an Internet service from its logical (anycast) address to the individual service nodes.

More specifically, versatile anycast presents an Internet service to its clients as a single fictitious MN. The anycast address X of that service then becomes the home address of that fictitious MN. The addresses of the service nodes, in turn, act as care-of addresses to which the traffic can be redirected. By disclosing different care-of addresses to different clients, versatile anycast can convince different clients that the MN has moved to different locations [see Figure 5.6]. Note that the client's higher (transport and application) layers retain the illusion that they communicate with the one and only node holding address X, as the translation between home and care-of addresses is confined in the network layer. This effectively enables the service nodes to jointly service their clients via a single anycast address, which allows for preserving the traditional service access model based on single contact addresses.

The following sections discuss how we implement the two phases of versatile anycast using Mobile IPv6. We first demonstrate how to implement the anycast address such that it is always reachable, yet it can also be moved between subsequent contact nodes. Then, we show how to implement transparent client handoffs between service nodes. Both these features together allow service nodes to join and leave the service infrastructure at will, thus enabling service adaptability. Note that the following sections assume that each service client supports MIPv6. We discuss how a service can handle clients that do not support MIPv6 in Section 5.6.1.

Anycast address implementation

To implement the first phase of versatile anycasting, one has to provide an anycast address and make sure that the traffic sent to that address ultimately reaches some node within the service infrastructure. A simple solution could be to choose the

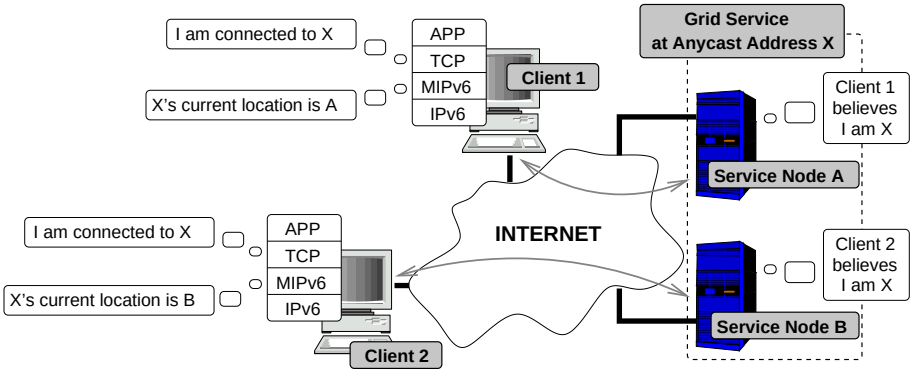


Figure 5.6: Communication with an Internet service

address of an arbitrary service node as the anycast address. In that case, however, the anycast address would be bound to this selected service node. This would make it impossible to contact the Internet service once that node has left the service infrastructure.

To circumvent this problem, a completely new address must be issued that is then used as the anycast address. Dynamically creating an address is not difficult, as any IPv6 node can produce addresses belonging to its own network. The service node which created the new address can then make the address reachable by attaching it to its network interface, and advertise it as the service address. Later on, if the service node decides to leave the service infrastructure, all that needs to be done is *move* the anycast address to any other service node that remains in the service infrastructure. We refer to the service node that holds the anycast address at a given moment as a *contact node*.

To enable the service to move its anycast address at will, the contact node performs a two-step procedure immediately after having created the anycast address. First, it establishes an *IPsec security association* for that address with its home agent. Recall that such an association is used by MIPv6 to authenticate mobile nodes to their home agents. It then forwards the association data and the HA's address to one or more *backup nodes* within the service infrastructure. Given that any node holding the association data is considered by the home agent to be the contact node, any backup node can now impersonate the contact node when communicating with the home agent. Note that throughout the entire service lifetime, the service appears to that home agent as a regular mobile node. The home agent therefore does not need to run any specialized software in addition to MIPv6.

The contact node and all its backup nodes form a fault-tolerant group, whose goal is to keep the anycast address persistent. This is achieved by enabling any backup node to take over the anycast address should the contact node leave the

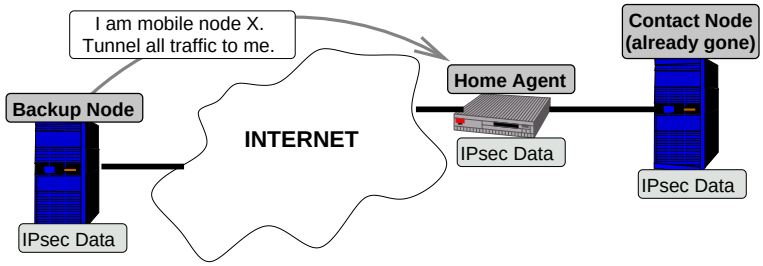


Figure 5.7: Taking over the anycast address

service infrastructure. Note that the contact node must trust its backup nodes that the address takeover does not take place as long as the contact node remains within the service infrastructure.

To take over the contact address, one of the backup nodes convinces the home agent that it is actually the contact node that has moved to another network. To this end, that backup node authenticates itself to the home agent using the IPsec data obtained from the contact node, and reports its address as the new care-of address of the contact node [see Figure 5.7]. This results in tunneling the traffic targeting the anycast address to the backup node through the home agent, which effectively turns the backup node into a new contact node. Doing so preserves the reachability of the anycast address as all the traffic addressed to the service keeps on reaching one of the service nodes. Note that some other backup node must take over the anycast address should the new contact node leave the service infrastructure.

Although the anycast address is now stable, service access performance might still turn out to be poor because extensive tunneling to the new contact node can overload the home agent and introduce communication latency. These limitations are addressed by route optimization wherein the care-of address of an MN is revealed to a CN. Given the care-of address, the MIPv6 layer of the CN transparently translates between home and care-of addresses of the mobile node.

Since an Internet service appears to its clients and home agents as a regular mobile node, it can also use route optimization. As a consequence, clients can communicate directly with the new contact node using its actual address. This is likely to result in better service access performance.

The remaining question is how to enable multiple service nodes to use the same anycast address simultaneously. So far, we have discussed how all the clients can directly communicate with only one service node, namely the contact node. The next section describes how the anycast address is effectively shared by enabling the contact node to transparently handoff its clients to other service nodes, which constitutes the second phase of versatile anycasting.

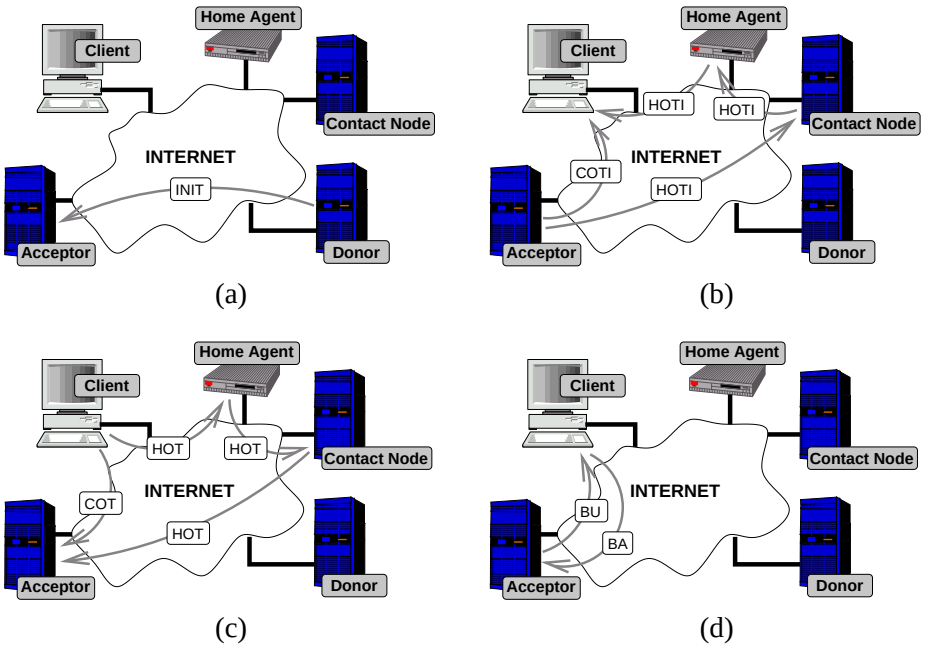


Figure 5.8: MIPv6 handoff

MIPv6 handoff

The implementation of the anycast address ensures that each client request reaches the contact node. However, this node should not process all the incoming requests by itself. It therefore needs a mechanism that enables it to transparently hand off each request to another service nodes, which later may themselves transparently hand it off again. We refer to the service node that hands off a client as a *donor*, and to the service node that takes over the client as an *acceptor*.

An important observation is that while handoffs must be transparent to the client application, they need not be transparent to the underlying layers of the client-side protocol stack. For example, the MIPv6 layer running at CNs hides the movements of MNs from the upper layers by translating home addresses into care-of addresses, and vice versa. We propose to exploit this address translation to implement client handoffs between service nodes.

Recall that the address translation in MIPv6 is performed according to bindings created during MIPv6 route optimization. As we discussed in the previous section, an Internet service already exploits this mechanism to establish direct communication between the contact node and the clients. However, since route

optimizations are performed separately for each client, the service can also use them to hand off individual clients between any pair of service nodes.

The goal of an MIPv6 handoff is to cause the client traffic sent to the anycast address to be redirected to the acceptor's address. This requires convincing the client that the service has just changed its care-of address to that of the acceptor, as only then will the client update its translation bindings accordingly. To this end, the service carefully mimics the signaling of a mobile node performing route optimization.

The MIPv6 handoff signaling is coordinated by the acceptor, but initiated by the donor, which sends a special *Init* message to the acceptor. That message contains the client address and the sequence number used during the previous route optimization [see Figure 5.8(a)].

Having received the *Init* message, the acceptor starts acting as an MN running MIPv6 route optimization. It first sends the HoTI and CoTI messages [see Figure 5.8(b)]. Note that the acceptor must tunnel the HoTI message to the contact node, which then tunnels it to the client through the HA.

The HoTI and CoTI messages cause their corresponding HoT and CoT messages to be sent by the client, which acts as an MIPv6 correspondent node during the MIPv6 handoff [see Figure 5.8(c)]. The HoT message is also tunneled twice, by the HA and by the contact node. This requires that the contact node maintains a list of pending MIPv6 handoffs.

Having received the HoT and CoT messages, the acceptor sends a BU message to the client, which updates its binding cache entries and acknowledges the update with the BA message [see Figure 5.8(d)]. From that moment on, the communication between the client and the service proceeds between the client and the acceptor.

The MIPv6 handoff enables the acceptor to communicate with the client on behalf of the service on the network level. However, Internet services commonly communicate with their clients by means of stateful connection-oriented protocols such as TCP. In that case, handing off a client at the network level alone is not enough as it would break the transport-level connections. The next sections discuss how to preserve such connections during a handoff.

5.4.3. Transport-level handoff

Many Internet services use TCP connections for client-server communication. In that case, redirecting the client's IP packets from the donor to the acceptor is not sufficient to enable the acceptor to communicate with that client. This is because maintaining a TCP connection requires that the client and the server maintain some connection state.

Preserving handoff transparency requires that apart from switching the client's IP traffic, this server-side connection state is also transferred from the donor to the acceptor. Transferring the TCP connection state from one node to another is commonly referred to as *TCP handoff*. Note that TCP handoff does not affect client-side state.

Performing a TCP handoff together with an MIPv6 handoff results in transparent switching of the complete TCP connection from the donor to the acceptor. As a result, the client and the acceptor communicate directly with each other, which eliminates the need for shared front ends often employed by clusters. This makes TCP handoffs implemented by an *adaptable* Internet service fundamentally different from those implemented by their cluster-based counterparts.

This section describes how TCP handoffs are supported in an adaptable Internet service. We first describe some basic properties of the TCP protocol, and then propose a procedure to hand off TCP connections on top of MIPv6 handoff.

TCP properties

TCP is a reliable communication protocol based on IP. Reliability of communication is ensured by means of acknowledgments and retransmissions. In TCP, each transmitted packet is numbered and must be acknowledged by the receiver. Should that not happen within some period of time, then the packet is assumed to be lost and therefore periodically retransmitted until its acknowledgment arrives, or a timeout occurs.

TCP requires the communicating parties to maintain some state. This state contains information about recent acknowledgments and packet (re)transmissions, along with buffers containing the data that have not yet been sent or acknowledged. The total size of a TCP connection state depends on the buffer sizes, and varies from 90 bytes to around 90 kB.

The control states maintained by both ends of a TCP connection must remain consistent for the protocol to function properly. If one party receives a message proving that the other end is not in a legal control state, then it *resets* the connection.

Each end of a TCP connection is attached to a TCP socket. Sockets are an abstraction of various communication mechanisms provided by the operating system. Client applications and service implementations use TCP sockets to send and receive data over TCP connections. Operating systems, in turn, use TCP sockets to store the state of these connections.

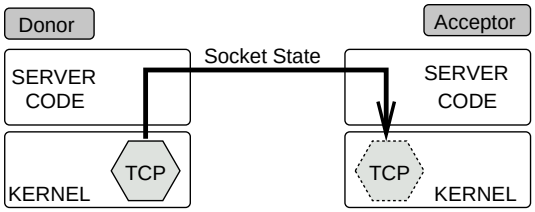


Figure 5.9: Socket migration

TCP handoff

Transferring the state of a TCP connection effectively means that the server-side TCP socket is migrated from the donor to the acceptor. To this end, the donor must extract the socket state from the operating system's kernel and send it to the acceptor. The acceptor, in turn, re-creates the socket in its own kernel based on the received state [see Figure 5.9].

We support TCP socket migration by means of the open-source TCPCP package [Almesberger, 2004]. It consists of a user-level library and a patch for the Linux kernel. TCPCP enables any donor application to extract an open TCP socket from the kernel in a serialized form. Given that serialized form, TCPCP re-creates the TCP socket in the acceptor application, possibly running on another node. The IP-level traffic associated with the TCP socket is assumed to be switched by some other mechanism.

The problem here is that while the socket is being migrated, the client may send data or acknowledgments to the server. We must therefore ensure that packets issued by the client during the migration can never reach a node that does not hold the corresponding TCP socket. Otherwise, the receiving member node would issue TCP control messages back reporting a missing socket, which would cause the connection to be reset. TCPCP solves this problem by maintaining two separate instances of the server-side socket during the period when it is unclear whether client-issued packets will reach the donor or the acceptor. In this way, the client traffic sent during the handoff always reaches some socket instance and can never trigger the connection reset.

Maintaining two server-side socket instances forces TCPCP to keep their states consistent with each other, and with the connection state held by the client. TCPCP achieves that by simply disallowing the TCP connection state to change during the migration. To this end, it freezes the socket right before extracting it from the kernel. A frozen socket does not send any data nor acknowledgments, and it silently drops all the incoming data or acknowledgments without processing them. Note that the dropped data and acknowledgments will be retransmitted by the client. The socket is unfrozen after the IP-level traffic has been switched.

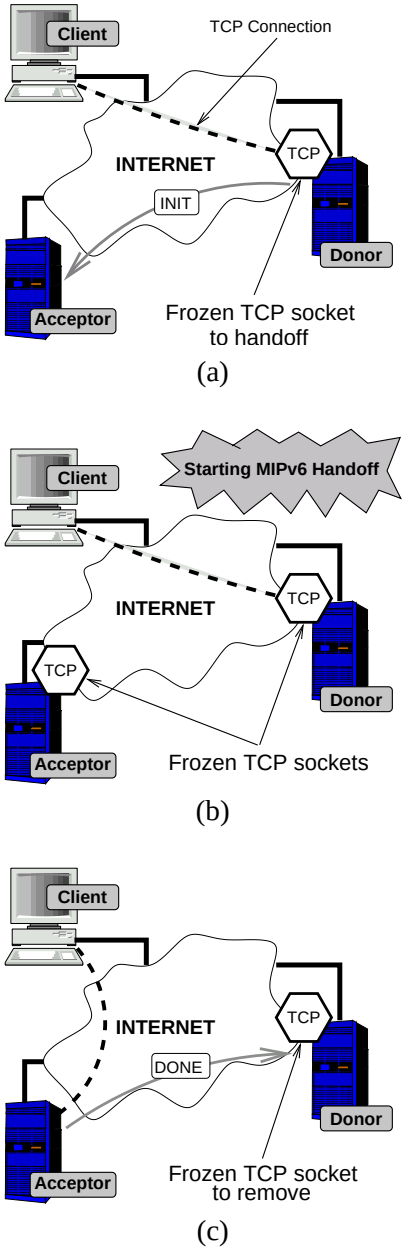


Figure 5.10: TCP handoff

The TCP handoff procedure is depicted in Figure 5.10. The donor first freezes and extracts the TCP socket from the kernel. The socket is then sent in the *Init* message (also used for MIPv6 handoff in Figure 5.8) to the acceptor, which re-creates the socket in its own kernel [Figure 5.10(a)]. Having re-created the socket, the acceptor conveys the MIPv6 handoff to switch the client traffic from the donor to the acceptor [Figure 5.10(b)]. Note that the two server-side socket instances are kept frozen during the MIPv6 handoff. Once the MIPv6 handoff has been completed, the acceptor unfreezes its socket, which can immediately be used to communicate with the client. The acceptor also notifies the donor about the handoff completion with a *Done* message, so that the donor can safely remove its frozen socket instance [Figure 5.10(c)].

Combining the TCP and MIPv6 handoffs allows an Internet service to migrate server-side TCP sockets among nodes within its infrastructure without breaking the associated TCP connections. To maintain the handoff transparency, however, the service must also ensure that the data sent over this connection by the acceptor are consistent at the application level with those sent by the donor before the handoff. We discuss this issue next.

5.4.4. Application-level handoff

Migrating the server-side TCP socket enables the acceptor to send service data to the client over the same TCP connection that was used by the donor before the migration. As a consequence, each socket migration logically divides the service response data sent to the client into two parts, depending on which service node actually sent the data.

Preserving the handoff transparency requires that this logical division remains invisible to the client, which expects all the response data to be sent by a single service node. The part sent after the handoff must therefore seamlessly match the part sent before the handoff, and all the parts together must form a response that is valid in terms of the service protocol.

Generating subsequent response parts without violating the service protocol requires that the donor passes the application-level state of the connection to the acceptor. Given that state, the acceptor generates and sends its response part as if it had generated all the previous response parts as well.

Passing the application-level state requires it to be serialized. The serialization method is typically application-specific. In HTTP, for example, a response is generated after receiving an HTTP request, and consists of a header and the actual requested content. In that case, the serialized application-level state consists of the HTTP request being serviced, an indicator saying whether the HTTP header has already been sent, and the description of the content part that has been sent so

far. If the content is a static document, then such a description can simply be the document name and the offset at which the previous content part ends.

The donor sends the serialized application-level state to the acceptor together with the *Init* message depicted in Figures 5.8 and 5.10. Recall that this message also contains all the data necessary to perform handoffs at the transport and network layers. Constructing such a message therefore requires that the donor concatenates the sequence number from the local MIPv6 implementation, the TCP socket state, and the application-level state.

To relieve the service implementation from dealing with handoffs at different levels, the construction of *Init* messages can be implemented in a separate library. We discuss the syntax of the library calls in C, but the library itself can also be implemented in Java or in any other language. The core function of that library is:

```
client_handoff(client_socket_fd, acceptor_IP, app_state)
```

which constructs the *Init* message, sends it to the acceptor, and waits for the *Done* message that signals the handoff completion. The donor would call this function to migrate a given client socket along with the application-level state to the acceptor. Once the call returns, the donor closes the client socket using the standard close() call.

All that is needed at the acceptor's side is to create a special socket bound to a well-known port, which is used to receive *Init* messages from donors. Whenever a message arrives at this socket, the acceptor can call another library function to accept an incoming handoff:

```
client_receive(special_socket_fd, &client_socket_fd, &app_state)
```

This function reads the *Init* message from the special socket, performs the MIPv6 and TCP handoff, and sends the *Done* message to the donor once these handoffs are complete. Finally, the function *returns* the client socket re-created by the TCP handoff and the application-level state. The service instance running on the acceptor simply needs to unserialize the received application-level state and determine what data should be sent to the client next. Once this is done, these data are transmitted over the re-created client socket just as over any other client socket created using traditional methods. However, the re-created socket must be closed using a special library function client_close(client_socket_fd), which ensures that the MIPv6 binding cache entry on the client side is deleted.

5.4.5. Summary

Versatile anycast provides Internet services with logical addresses, and enables each such service to redirect client traffic from its logical addresses to the physical

addresses of the service nodes for load balancing. The service nodes can also handoff individual each client among themselves, which enables service nodes to join and leave the service infrastructure as necessary. The resulting decoupling between the logical service address and the service infrastructure contributes to the improved service adaptability and comes at the expense of upgrading the service nodes such that they support versatile anycast functions.

The complexity of upgrading service nodes depends on the details of MIPv6 implementation specific to the operating systems running on these nodes. For example, the MIPv6 implementation that we used in our experiments consists of a user-space daemon that manages kernel-level packet translation mappings. These translation mappings are relatively generic, and most likely will soon be included in standard kernel releases. Modifying the MIPv6 daemon, in turn, is relatively simple, as it is a regular user-level program. On the other hand, TCP socket extraction and injection is not among standard features provided by most operating systems, and must therefore be added separately. Whether or not it poses a problem, depends on how proficient the service operators are with installing kernel extensions. Furthermore, if such functionality turned out to be commonly used, then it could also be included in standard kernel releases at some point. Also, note that implementing TCP handoff in a local-area network traditionally requires deploying a specialized router dispatching client requests among service nodes [Andreolini et al., 2003a, b]. This is not necessary in the case of distributed handoffs based on Mobile IPv6, as they employ *unmodified* MIPv6 home agents to implement stable service addresses and to perform traffic switching.

Finally, note that versatile anycast is just a routing mechanism and as such cannot make an Internet service adaptable by itself. To this end, the service needs to implement a number of additional functions such as membership management and load balancing. These functions enable the service to react to the changes in the composition of its infrastructure, and to select service nodes to which clients should be redirected or handed off. Both membership management and load balancing can be implemented using standard techniques [Voulgaris et al., 2005; Cardellini et al., 2003].

5.5. EVALUATION

We evaluate the performance of versatile anycast using a simple testbed [see Figure 5.11]. The core of that testbed is a NISTnet router, which connects the client machine to a service infrastructure [Carson and Santay, 2003]. The infrastructure consists of two service nodes located in different networks, which are connected to the NISTnet core via their home agents.

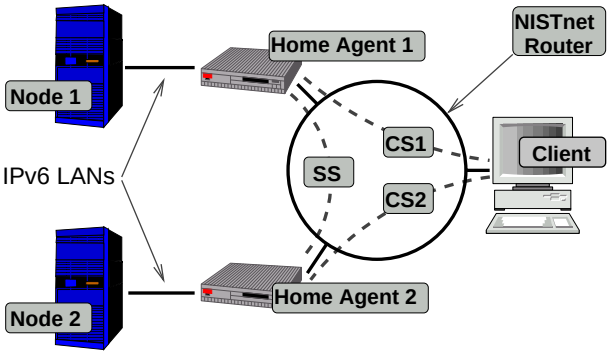


Figure 5.11: Topology of the versatile anycast testbed

We use the NISTnet router to emulate wide-area latencies. However, since NISTnet is not IPv6-enabled, we established three IP6-in-IP4 tunnels: *SS* to control packet transmission between the member nodes, and *CS1* and *CS2* to control packet transmission between these member nodes and the client.

The NISTnet router runs Linux 2.4.20. All the remaining machines run Linux 2.6.8.1 and MIPL-2.0-RC1, which is an open-source MIPv6 implementation for Linux [MIPL, 2006]. All the machines are equipped with PIII processors, with clocks varying from 450 to 700 MHz.

5.5.1. Server access latency

The anycast address implementation based on tunneling provided by MIPv6 causes the client packets to be routed through the home agent, which then tunnels them to the contact node. The service access latency therefore consists of two parts: the latency between the client and the home agent, and the latency between the home agent and the contact node.

To verify whether this is indeed the case, we developed a simple UDP-echo application. A UDP-echo client sends a 128-byte UDP packet to the service, which sends that packet back. The client measures the round-trip time as the delay between sending and receiving the packet.

We used two different configurations of the UDP-echo service. Both configurations use the anycast addresses created by Node 1. However, whereas Node 1 belongs to the service in the first configuration, it does not in the second one. In that case, the packets are tunneled between Home Agent 1 and Node 2.

For each service configuration, we have configured NISTnet with several combinations of latency values. Packets transmitted through the SS tunnel were delayed by various latencies Lat_{SS}. Packets transmitted through the CS1 tunnel, in turn, were delayed by various latencies Lat_{CS1}. For each pair of latencies, we it-

Table 5.1: Handoff time decomposition (without NISTnet delays)

No.	Operation Name	Inter-node Bandwidth			
		100 Mbps	2 Mbps	1.5 Mbps	1 Mbps
1	Socket Extraction	0.8 ms	5.8 ms	6.9 ms	11.8 ms
2	State Transfer	6.5 ms	319.1 ms	434.1 ms	648.2 ms
3	Socket Re-creation	2.2 ms	2.1 ms	2.1 ms	2.2 ms
4	Return-Routability Proc.	2.5 ms	3.7 ms	4.9 ms	8.9 ms
5	BU-Message Construction	2.7 ms	2.7 ms	2.7 ms	2.7 ms
6	Binding-Management Proc.	2.6 ms	2.6 ms	2.6 ms	2.6 ms
7	Socket Activation	1.1 ms	1.1 ms	1.1 ms	1.1 ms
Total Time:		18.4 ms	337.1 ms	454.4 ms	677.5 ms

eratively ran the UDP-echo client 100 times and calculated the average over the reported round-trip times.

The results were very consistent. The average reported round-trip time was $2 * Lat_{CS1} + X$ for configuration 1, and $2 * Lat_{CS1} + 2 * Lat_{SS} + Y$ for configuration 2, where X and Y are small additional delays (on average 2.13 ms and 3.61 ms, respectively). We attribute the X and Y delays to the latency of Ethernet links and the time of local processing at all the machines visited by the UDP packets.

Recall that the Internet service can use route optimization to enable direct communication between its clients and the contact node. However, since route optimization takes place in parallel to the application-level communication, we do not consider it in this experiment, and analyze it only when evaluating the handoff times below.

5.5.2. Handoff time decomposition

Versatile anycast enables the service nodes to hand off a client TCP connection among each other. In this experiment, we investigate how much time is necessary to hand off a TCP connection, and what operations consume most of that time.

Handoffs are performed by a simple service that delivers 1 MB of content upon request. The client first opens a TCP connection to Node 1 acting as the contact node. Node 1 transfers 500 kB of data, and hands off the connection to Node 2 immediately after the last send() call returns. Node 2 sends another 500 kB of data and closes the connection.

The total handoff time can be divided into seven phases [see Table 5.1]. The phases are delimited by the event of sending or receiving some specific packets, which we time-stamp to mark the boundary between subsequent phases. To detect events, we monitor all the packets exchanged in the testbed using *tcpdump* listening on all the network interfaces of the NISTnet router.

Table 5.1 reports the delays averaged over 100 download sessions. We have emulated various speeds of the upstream DSL connections by shaping the traffic sent from the home agents to the NISTnet router using the standard *cbq* queuing discipline available in the Linux kernel. The results for unshaped 100 Mbps Ethernet are included for completion.

As can be observed, extracting the socket at the donor apparently takes between 0.8 and 11.8 ms depending on the network bandwidth (Phase 1). However, since this operation is entirely local, it should not depend on the bandwidth at all. We have therefore verified these results by measuring the actual time spent in the socket-extracting call, which turned out to be 0.8 ms on average. We believe that the higher values obtained using packet monitoring result from transmission delays introduced by bandwidth shaping.

Most of the total handoff time is spent on transferring the socket state (Phase 2). The duration of this phase is proportional to the network bandwidth, as each time the donor transfers the 90 kB of the socket state to the acceptor. This time accounts for up to 95% of the total handoff time when emulating 1 Mbps DSL lines.

Local phases such as re-creating the socket, constructing the BU message, and activating the socket correspond turn out to be relatively fast and independent of the bandwidth (Phases 3, 5, and 7). The return-routability procedure, in turn, demonstrated some dependency on the bandwidth (Phase 4). However, since the packets transmitted during this phase are very small, we believe that this dependency is artificial, and results from delaying packets by the shaping mechanism previously observed for Phase 1.

Interestingly, the artificial delays introduced by traffic shaping cannot be observed for the binding management procedure, where the BU and BA messages are exchanged between the acceptor and the client (Phase 6). This is probably because the low network activity during Phases 3-5 causes the state of the shaping mechanism to be reset by the time Phase 6 starts, which enables the two packets to be transmitted without any delay.

We also performed the same experiment for various combinations of L_{SS}, L_{CS1}, and L_{CS2} latencies emulated by NISTnet (we used $L_{CS1} = L_{CS2}$). The results are similar to those presented in Table 5.1, except that the time spent in some phases varies proportionally to the NISTnet latencies. In particular, phase 2 varies by L_{SS}, phase 4 varies by $2 * L_{SS} + 2 * L_{CS1}$, and phase 6 varies by $2 * L_{CS2}$. The additional delays correspond to the latencies of network paths followed by the messages exchanged during the respective phases. Note that should any of the MIPv6 packets be lost, it will be automatically retransmitted; in that case, the overall handoff time will obviously be extended by the MIPv6 retransmission timeout of 1 second.

5.5.3. State transfer optimization

The previous experiment shows that most of the handoff time is spent transferring the socket state from the donor to the acceptor. The reason why that transfer takes so long is that in this experiment the donor extracts the socket immediately after the last send() call returns. This means that the socket buffers are nearly full, which results in the socket size taking about 90 kB.

One way of reducing this size is to simply wait for some time as the donor gradually sends the data stored in the socket buffers and removes the data acknowledged by the client from the buffers. This would allow the client to receive and acknowledge at least some of the data, which in turn would reduce the socket state. In this experiment, we investigate how such waiting affects the handoff time.

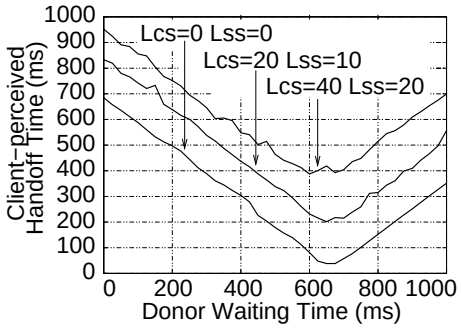
We modified our server so that it would wait for a given period of time between passing the last data to the socket and starting the actual handoff procedure. We also modified the client such that it measures its perceived handoff time. We define the client-perceived handoff time as the delay between receiving the last packet from the donor and the first packet from the acceptor.

Given the modified application, we repeatedly ran 100 download sessions for 1 MB of content and waiting times varying from 0 to 1000 ms with a step of 25 ms. Similar to the previous experiments, we emulated three different DSL connection bandwidths and various combinations of wide-area latencies. The results are presented in Figure 5.12.

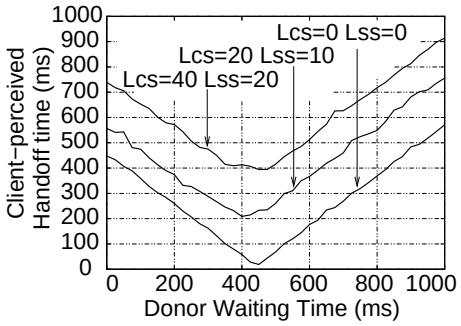
Increasing the donor's waiting time causes the client-perceived handoff time to decrease to some minimum value. Having reached that value, the client-perceived handoff time starts increasing. We verified that the minimum value corresponds to the situation when the socket was extracted right after receiving the last acknowledgment from the client, which removes the last packet from the socket buffers. As a consequence, the socket state has only 90 bytes, which can be transferred in the time of the one-way latency between the donor and the acceptor. This eliminates the delay resulting from transferring a large socket state over a low-bandwidth connection. We conclude that the donor should always empty its output TCP buffers before freezing the socket and starting the handoff.

5.5.4. Handoff time optimization

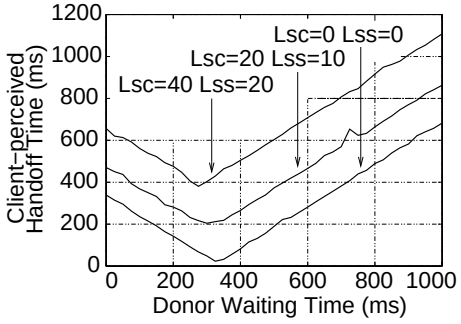
Now that the socket state is reduced to sending a single packet from the donor to the acceptor, and given that the local processing times are negligible, the actual handoff time depends only on the latencies of the paths followed by the messages exchanged during the handoff. In this experiment, we investigate whether this time can be reduced even further.



(a) 1 Mbps



(b) 1.5 Mbps



(c) 2 Mbps

Figure 5.12: Client-perceived handoff times for various upstream node connection bandwidths

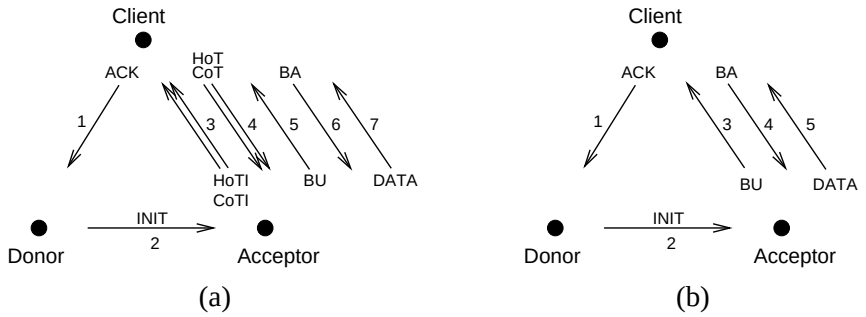


Figure 5.13: Optimization of wide-area latencies

Recall that the client-perceived handoff time is the delay between receiving the last packet from the donor and the first packet from the acceptor. The beginning of the client-perceived handoff time corresponds to sending the last acknowledgment to the donor [see Message 1 in Figure 5.13(a)]. Upon receiving that acknowledgment, the donor sends the *Init* message to the acceptor, which then runs the complete MIPv6 handoff. Once the MIPv6 handoff is complete, the acceptor sends the first packet containing the application data to the client. The client-perceived handoff time ends once that packet is received by the client.

In fact, all these steps do not need to be performed sequentially. In particular, the acceptor can run the return-routability procedure in advance while the donor is still busy with transferring data to the client, as no data other than the client address is needed for that. Performing the return-routability procedure in advance eliminates its time from the client-perceived handoff time, and allows the donor to send the BU message immediately after the *Init* message arrives [see Figure 5.13(b)].

To allow the acceptor to run the return-routability procedure in advance, the donor must notify the acceptor about the upcoming handoff by sending a *Prepare* message containing the client address. This message is sent when the donor has passed all its data to the socket and is about to start waiting for the last acknowledgment from the client. Having received the *Prepare* message, the acceptor conveys the return-routability procedure and keeps the resulting Kbm so that it can be sent in the BU message once the *Init* message arrives.

To investigate the impact of performing the return-routability procedure in advance, we modified our test application once again. In the new version, the donor sends the *Prepare* message immediately after returning from the last send() call, and then waits for the socket to become empty. The acceptor performs the return-routability procedure upon receiving the *Prepare* message, and waits for the *Init* message before sending the BU message to the client.

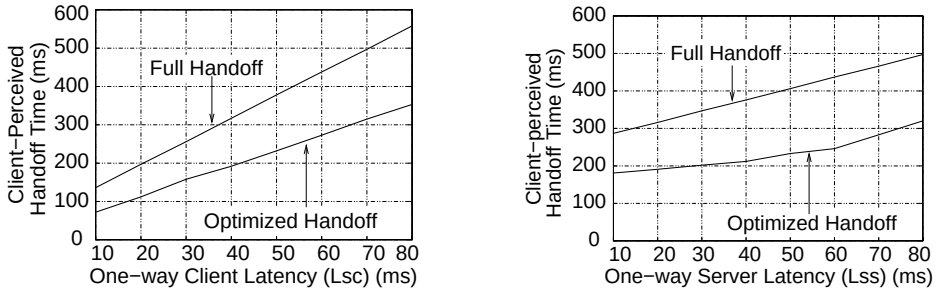


Figure 5.14: The impact of performing return-routability procedures in advance: with fixed L_{SS} (a), and with fixed L_{CS} (b)

Similar to the previous experiment, we measured the average client-observed handoff times for 100 download sessions with the network bandwidth shaped to 2 Mbps and various combinations of L_{CS} and L_{SS} latencies. Figure 5.14(a) shows the results obtained for L_{SS} fixed to 20 ms and L_{CS} varying from 10 to 80 ms, whereas Figure 5.14(b) shows the results obtained for L_{CS} fixed to 40 ms and L_{SS} varying from 10 to 80 ms.

As can be observed, performing return-routability procedures in advance results in the reduction of client-observed handoff times. The reduction is proportional both to the latency between the client and the server and to the latency between the member nodes. This is because our optimization effectively reduces the client-observed handoff time from approximately $6 * L_{CS} + 3 * L_{SS}$ to approximately $4 * L_{CS} + L_{SS}$ since the time of tunneling the HoT/HoTI messages between the acceptor and the contact node is about L_{SS}. Note that the gain is lower if the donor's waiting time is too short to allow the acceptor to complete the return-routability procedure before the *Init* message is sent. This can sometimes be observed for large L_{SS} values, which results in an increased slope in Figure 5.14(b) for L_{SS} equal or greater than 60 ms.

5.6. DISCUSSION

5.6.1. Client-side MIPv6 support

Our proposed mechanisms assume that client-side operating systems support the functionality of an MIPv6 correspondent node. This is already true for many popular operating systems, including Linux [MIPL, 2006] and Windows [MIPv6-SRL, 2006]. However, it might still happen that some potential service clients do not support MIPv6.

An Internet service running versatile anycast can support a small number of MIPv6-disabled clients. Recall that the client-side MIPv6 support is necessary to hand off clients among member nodes using route optimization, but it is not required to access the contact node. MIPv6-disabled clients can therefore be supported by tunneling all their traffic through the contact node. However, the number of MIPv6-disabled clients that are serviced simultaneously by the contact node should not be too large to prevent the contact node from becoming a bottleneck.

5.6.2. Multiple contact addresses

Although Internet services running versatile anycast normally have a single contact node, they can also create multiple contact nodes so that the effort of forwarding requests and handling non-MIPv6 clients is spread over several service nodes. In that case, each contact node has its own anycast address, which is advertised along with the other anycast addresses. Similar to what happens in the single-address scenario, each contact node in the multiple-address scenario must have a number of backup nodes. To keep the number of trusted nodes in the server low, each contact node may decide to select the remaining contact nodes as backup nodes. As a result, all the contact nodes form a fault-tolerant group wherein all the nodes can impersonate each other as necessary.

The multiple contact addresses must somehow be advertised to the clients. To this end, an Internet service may register them in the DNS. Note that this solution is significantly different from simple DNS redirection, as the DNS entries referring to logical addresses remain extremely stable even though the composition of the service infrastructure changes dynamically. Also, registering logical addresses in the DNS enables the Internet service to occasionally update the set of contact nodes so that it can stop using the home agents of former contact nodes soon after their provided addresses have been removed from the DNS.

5.6.3. Multiple client connections

Certain services might allow a client to simultaneously open multiple TCP connections to the same service, for example, to retrieve different parts of the service response in parallel. However, opening multiple TCP connections to an Internet service running versatile anycast via a single anycast address can lead to problems when the server decides to hand off any of these connections. Recall that the MIPv6 handoff updates the translation bindings maintained by the client's MIPv6 layer. However, since MIPv6 translation affects *all* the traffic between the client and the anycast address, either all the connections of a given client must be handed off simultaneously to the same acceptor, or none at all.

This limitation can be alleviated if the service has multiple anycast addresses. As each translation binding is associated with only one anycast address, it does not affect the traffic sent to other addresses. Provided that the client-side application opens simultaneous connections to different anycast addresses, the service can hand off each of them just like non-parallel connections. Note that handing off parallel connections to different service nodes effectively implements a parallel download from a distributed group of nodes, which has been shown to dramatically improve the client experience [Rodriguez et al., 2000].

5.6.4. Ungraceful node departures

In a large-scale service deployment, any node can leave the service infrastructure ungracefully, for example because of a hardware failure. In that case, it is too late to transfer the application- and transport-level state of client connections serviced by that node to some other node. Although MIPv6 enables another node in the server to intercept the client traffic related to these connections, they can no longer be serviced without the state information, and the service is forced to close them. Such unexpected connection closing may result in the service appearing to be unreliable.

This problem can be alleviated by instructing each service node to replicate the state of all its connections across a small number of other service nodes. Should a service node leave ungracefully, the service can try to recover the connections based on the replicated state. Note that the service's ability to continue servicing a given connection greatly depends on the state of that connection. For example, it might be impossible to recover the data that have been received and acknowledged by the service node's TCP layer after the replicated connection state was updated for the last time. This is because the service cannot force the client to retransmit the already-acknowledged data. Also, recovering from ungraceful departures tends to be application-specific, as applications themselves might provide some degree of resilience to sudden service outages. In the current server implementation, the node taking over the clients after an ungraceful departure simply closes the connections whose state it does not know. We treat more advanced strategies for connection recovering as an interesting topic for future research.

5.6.5. Global client redirection

Given the good properties of versatile anycast, one may wonder to what extent it can be applied to redirecting clients among a globally-distributed set of replicas. From the perspective of the anycast mechanism itself, there is little difference whether an anycast group consists of 10, 100, or 10,000 nodes, or whether these nodes are co-located or scattered over a wide-area network. This is because

switching always involves only a small number of nodes (the donor, the acceptor, and the contact node), and each of them can be located anywhere in the Internet.

The scalability of versatile anycast makes it tempting to build a globally distributed system as a single distributed server that exploits MIPv6 switching as the sole mechanism for client redirection. However, switching clients between server nodes requires that all the nodes involved are continuously monitored for availability and load. Meanwhile, monitoring a large number of widely distributed nodes is likely to introduce a significant communication overhead, resulting in a higher utilization of potentially scarce network resources. A globally distributed system should therefore restrict both the number and the dispersion of nodes falling into a single anycast group to avoid high operation costs.

Restricting the distribution and number of nodes means constructing distributed servers out of a small number of nodes that are relatively close to each other. For example, in a latency-driven system exploiting our placement algorithm discussed in Chapter 4, a single distributed server can be constructed out of nodes within a single network region. Given that all these nodes are proximal to each other in terms of latency, they can monitor the state of each other at a very low cost. The additional benefit from preserving low network latency within a distributed server is that it accelerates the communication between service components running on different server nodes, resulting in streamlined service operation.

Apart from implementing the switching mechanism, each distributed server must also be able to decide when each client should be switched and to which node. We believe that these decisions essentially depend on the service implementation. For example, if the service runs on many nodes performing identical functions, then it is possible to select among them based on classical metrics such as available bandwidth or CPU load. Otherwise, node selection might also be influenced by the functionality of each node or any other service-specific factors. For example, if different nodes are responsible for handling client requests of different types, then such a constraint must be considered during node selection. We expect that the growing complexity of service implementations will cause the ultimate node selection algorithm to increasingly often depend on service-specific factors.

Limiting the span of a distributed server to a single network region makes it necessary to employ another technique for redirecting clients to servers in different network regions. We thus obtain a two-stage redirection system: clients are first redirected to a nearby distributed server, which then hands them off to its individual nodes.

Recall that each distributed server aggregates any resources contributed by its member nodes, including, for example, disk space or network bandwidth. This enables each distributed server to autonomously manage many resources for a

large number of clients. However, as each distributed server consists of nodes proximal in terms of latency, it cannot optimize latencies to its clients, as latencies between all these nodes and a given client are very similar. On the other hand, latency is the primary factor that a latency-driven system strives to minimize. Such a system must therefore optimize latencies by means of mechanisms external to distributed servers, for example, when selecting the best distributed server for each client.

Latency-driven server selection can easily be achieved based on latency models produced by our latency estimation techniques described in Chapter 3. Given these models, the system first estimates the latencies between the client being redirected and all the distributed servers, and then selects the server whose estimated latency to that client is the shortest.

Once the server has been determined, its address needs to be passed to the client by some redirection mechanism. A crucial observation at this point is that the addresses of distributed servers are extremely stable, and can therefore be treated as regular addresses belonging to highly available nodes. As a consequence, the system can employ any scalable mechanism for redirection within static collections of nodes.

A good mechanism for redirecting clients among distributed servers is DNS redirection. As we discussed in Section 5.3, long-term address caching makes DNS unattractive when redirecting clients between volatile addresses of nodes forming distributed servers. However, when redirecting clients to long-lived addresses, DNS is the *de facto* standard employed by many highly successful distributed systems [Afergan et al., 2005; Barroso et al., 2003].

DNS redirection requires that the service runs a customized DNS server implementing the latency-driven server selection algorithm. In theory, this makes it necessary to deploy such a DNS server outside the service, preferably on some highly available machine. In practice, however, one can deploy a DNS server along with the service code on one of the distributed servers, which will then be responsible for both replica hosting and client redirection. Furthermore, the customized DNS server can be incorporated into the service in order to integrate the redirecting component with those responsible for latency modeling and replica placement. Such integration enables the redirector to, for example, use up-to-date latency information stored inside the latency modeler. We have demonstrated how to incorporate a DNS-based redirector into a peer-to-peer content delivery network in our previous study [Szymaniak et al., 2003].

5.7. CONCLUSION

We have presented versatile anycast, which allows for organizing a group of nodes into a distributed server that appears to its clients as a single node. Versatile anycast enables an Internet service deployed on such a group of nodes to preserve the traditional service access model, in which clients send their requests to the single (logical) address of the service. At the same time, the service address is not bound to any physical node, which allows for changing the server composition at any moment, for example, to adapt the server to evolving network conditions or new functional requirements.

Versatile anycast presents an Internet service to its clients as a mobile node. This enables the service to decouple its logical address from the addresses of the nodes forming the distributed server. Such a decoupling allows the service to dynamically map its logical address to any node while preserving the service reachability. By carefully re-mapping the logical address among its nodes, the service might ensure its continuous availability as long as at least one of its nodes is running.

The mappings between the logical and physical addresses can be changed on a per-client basis. This enables the service to transparently hand off clients among the service nodes at the network level while preserving optimal routing between the clients and the service nodes. The resulting distribution of client-servicing effort among multiple nodes effectively leads to aggregation of bandwidth provided by each node forming the distributed server, which can then meet higher bandwidth requirements than each of the nodes alone.

We have demonstrated that the overhead of contacting an Internet service via its anycast address can be estimated as the latency between the contact node and the home agent responsible for the contact address. The client-perceived handoff time has also been shown to be a linear function of the latencies among the client and the service nodes participating in the handoff. In our opinion, the cost of running versatile anycast is very low compared to the benefit of aggregating widely distributed resources without using any frontends.

We believe that versatile anycast has the potential to enable constructing highly available and powerful distributed servers in network regions where no single physical node is suitable for replica hosting. However, as routing of client requests can rely on versatile anycast only within distributed servers, another mechanism is necessary to redirect clients to and between distributed servers. We note that versatile anycast makes the addresses of distributed servers extremely stable, and therefore propose to inform clients about these addresses by means of classical DNS redirection.

As a routing mechanism, versatile anycast cannot make the distributed server fully operational by itself. To this end, an Internet service must combine versatile anycast with some techniques for membership management and load balancing. These techniques enable the service to decide when to use each of the functions provided by versatile anycast. We discuss this issue in the next chapter, when demonstrating how to incorporate all the techniques presented in this thesis into a single latency-driven system.

CHAPTER 6

Conclusion

Developing a large-scale distributed system is a complex task. The design of such a system needs to address all the problems resulting from the global distribution and the large number of nodes by which it is formed. Some of the problems are related to communication between nodes, such as long latencies, limited bandwidth, or unreliable network connections. Other problems are simply caused by the huge collection of nodes that needs to be dealt with. For example, the composition of truly large-scale systems changes continuously as individual nodes may join or leave the system at any moment. In such a situation, it is infeasible to even count the number of nodes in the system at any given moment, not to mention monitoring all of them, or collecting the complete information about the resources that each of them can provide.

The task of developing a latency-driven system is even more difficult, as such a system should rely only on solutions that take network latencies into account. Furthermore, it should also specifically employ techniques to minimize latencies between individual system components. Various aspects of large-scale distributed systems have been investigated for many years, but the issue of latency minimization in such systems has relatively seldom been addressed. Some proposed techniques try to minimize latencies by optimizing other metrics, such as the number of hops or the geographic distance. Other solutions argue that latencies can be minimized by means of overprovisioning bandwidth such that no queuing delays are introduced at intermediate routers. A real breakthrough took place when GNP made it possible to estimate latencies in large-scale systems using the concept of network positioning. However, the sole ability to estimate is not enough to develop an entire system that minimizes network latencies between its individual components.

Research contributions

This thesis has proposed a number of techniques applicable at subsequent steps towards designing a large-scale globally distributed system that minimizes network latencies between its components. Clearly, minimizing latencies requires some means of estimating them. The first step is therefore to implement a scalable latency estimation scheme. Believing that the concept of network positioning is the right approach to this problem, we have investigated how to implement GNP in a real-world environment.

The results are described in Chapter 3, which discusses our implementation of GNP incorporated into the large-scale infrastructure of the Google Web search engine. We demonstrated that GNP can be employed to estimate latencies between millions of Google clients, and proposed how to improve the latency models produced by GNP to increase their stability and hence reduce the frequency of model recomputation. We also evaluated the performance of client redirection based on latency models produced by GNP and discussed how latency information collected by a world-scale system can be used as a basis for a public, Internet-wide latency estimation system available to any distributed application. Finally, we also identified a number of system types that would not benefit from a third-party latency estimation service, and proposed that they perform latency estimation on their own in a federated manner.

Once latencies between nodes are known, the second step is to choose locations for various system components in a latency-aware manner. Many systems ignore latencies during replica placement, and take them into account only during client redirection. In that case, however, client redirection effectively attempts to compensate for the shortcomings of such replica placements. This causes the system to invest its efforts into solving problems that it in fact created itself by misplacing the replicas. We propose to avoid such situations by optimizing latencies during both replica placement and client redirection. Note that this is a practical consequence of our observation from Chapter 2, where we noted that replica placement and client redirection complement one another as each of them implements the assumptions made while addressing the other.

When solving the replica placement problem, we assume that each client is redirected to a replica proximal to that client in terms of latency. The replica locations should therefore be selected such that the latencies between clients and their closest replicas are minimized. The advantage of this approach is that it enables the system to provide statistical latency guarantees. For example, the system might place replicas such that 90% of clients are located within 50 milliseconds from the locations of their proximal replicas, and then redirect clients so that the placement properties are fully exploited.

Our solution to the problem of latency-driven replica placement is the HotZone algorithm described in Chapter 4. HotZone identifies network regions in which replicas should be located in order to minimize client-replica latencies. We evaluated the performance of HotZone in terms of both client-replica latency and computation times, showing that it produces high quality placements within times several orders of magnitude shorter than its earlier counterparts. HotZone performs its computations faster by exploiting the geometric properties of latency models produced by GNP. The resulting low computational complexity of HotZone makes it particularly useful in situations where the number of candidate replica locations is very large, for example, when deciding how to place content on a large number of replica servers. However, HotZone can also be used to produce server placements. The difference between these two cases is that server placement tends to depend solely on client locations, whereas placing the replicas requires that both client locations and client access patterns are taken into account. Each of these cases translates into a different definition of a network region, and HotZone can be configured to use either definition depending on the placement type that needs to be produced.

Once network regions for replica placement have been identified, the third step is to deploy actual replicas on nodes within these network regions. As the ability to host a replica might depend on certain requirements such as disk space and available network bandwidth, one possible solution could consist of inspecting nodes one by one until a candidate meeting all the requirements is found. However, finding such a node might sometimes be infeasible, for example, when all the nodes in a given region have relatively low performance. Also, even when a powerful-enough node can be found, it might turn out to be unreliable, forcing the system to move the replica to some other node.

We believe that solving performance or availability problems faced by nodes within a given network region should not involve nodes from other network regions. One reason for that is simply to avoid the need for inter-region communication that is inherently slower than its intra-region counterpart. More importantly, however, confining the problem-solving process to a network region enables one to treat all the hosts in a network region as a single entity responsible for solving its own performance problems with performance or availability. This, in turn, conceptually brings a network region close to a powerful and highly available distributed server. We envisage that an Internet service can exploit such servers as reliable high-performance hosting facilities for its replicas, even though each server consists of potentially slow and unreliable nodes.

Constructing a distributed server out of nodes potentially scattered over a wide-area network requires some means of preserving the server distribution transparent to the clients. Chapter 5 proposed to achieve distribution transparency using

versatile anycast, which employs Mobile IPv6 to organize all nodes forming a distributed server into a single anycast group addressed with its anycast address. Similar to other anycast implementations, versatile anycast enables all nodes within an anycast group to jointly service client traffic heading to the anycast address of that group. In contrast to earlier implementations, however, versatile anycast leaves each anycast group in full charge of how the client traffic is split among the nodes within that group. At the same time, versatile anycast preserves optimal routing between these nodes and their clients, and requires no modifications to client-side software.

We developed a prototype implementation of versatile anycast, and demonstrated that it enables widely distributed nodes to share the same address while communicating with their clients. Also, we showed how to combine versatile anycast with software for TCP socket serialization, and how such a combination enables a distributed server to transparently handoff client TCP connections between nodes scattered over a wide-area network. Finally, we discussed how the logical nature of anycast addresses and their ability to move between server nodes allows for ensuring high availability of a distributed server despite potentially frequent changes in its composition.

While the internal organization of a distributed server very much depends on the particular application it is running, the external communication with such a server via its anycast address is relatively straightforward. However, because a latency-driven system will in most cases consist of many distributed servers, it must provide some redirection scheme so that client requests are distributed among these servers in a latency-aware manner. As discussed in Chapter 5, we propose to select distributed servers based on latency models produced by GNP, which enables one to quickly identify the distributed server likely to offer the lowest latency to a given client. Such latency-aware server selection accomplishes a situation we previously assumed for proximity-based request routing when placing replicas using HotZone. We assume that the server selection algorithm runs on a customized DNS server run by the system, which employs the global DNS infrastructure to announce the addresses of distributed servers to the clients.

Future work

The number of issues related to latency-driven replication in globally distributed systems is very large. It is therefore impossible to address all of them in a single dissertation. To this end, the research presented in this thesis can be continued in several different directions.

First of all, deploying various applications on distributed servers requires that nodes forming these servers cooperate in order to jointly service client requests. Although the details of such cooperation are likely to be application-specific, there

exist numerous generic techniques that can be potentially useful at this point, such as maintaining redundant application-level state on multiple nodes (for quick recovery from node departures [Ramabhadran and Pasquale, 2006]), running multiple instances of internal services according to client demands (for higher service performance and load balancing [Karve et al., 2006]), mirroring TCP connection state on several nodes (for improved distribution transparency [Zhang et al., 2004]), or switching between replication policies according to changes in client access patterns (for more efficient resource utilization [Pierre et al., 2001]). One of the future research topics could therefore be to focus on how exactly such techniques can be abstracted as separate services and combined with our solutions to be used by different applications.

Another interesting research question is what application-specific techniques must be incorporated into a distributed server running a given application. Such techniques are likely to include custom intra-server redirection policies, and distributed state management schemes. For example, in a peer-to-peer environment, distributed servers could act as reliable and high-performance hosting facilities for BitTorrent trackers [Cohen, 2006]. Running a distributed tracker shall most likely require some specialized techniques for mapping incoming peer connections to individual server nodes, for example, based on the node availability and the number of connections serviced by each node. Similarly, all the nodes will need to exchange information about chunks available at different peers. The deployment of any particular application on a distributed server can therefore be perceived as a separate research area.

Apart from investigating various aspects of distributed server design, one might also look into global system organization. For example, one could construct a number of distributed servers to run a content delivery network [Pierre and van Steen, 2006]. Such a CDN could then employ a variety of techniques know from regular CDNs to manage its content. However, in order to remain truly latency driven, it would rather focus on quickly adjusting its operation to changes in network latencies. To this end, it should integrate the components responsible for latency estimation, replica placement, and client redirection in order to minimize the delay between noticing a change in network latency and reacting to that change by, for example, updating the replica placement or adjusting the redirection policy [Sivasubramanian et al., 2004]. Yet another research question is therefore how exactly such integration should be performed, and what reaction to changes in network latencies leads to biggest improvements in system performance.

In particular, note that updating the replica placement in response to changes in network latencies can have a different scope, as the system can decide to move either all the replicas, only a subset of them, or none at all. Whether the chosen scope is right depends not only on network latencies, but also on the cost of in-

stalling new replicas, which can be expressed, for example, in terms of network bandwidth utilized for that purpose. We therefore expect that there exist a trade-off between optimizing client-replica latencies by frequent replica relocation, and minimizing the system maintenance cost by limiting the number of such relocations. Investigating this trade-off and devising exact strategies for replica creation and relocation clearly requires more research.

Lessons learned

Our research into latency-driven replication for globally distributed systems allows for drawing two major conclusions. First, when designing a globally distributed system, one must take network latencies into account, even though their importance might be uneasy to notice at the early stages of system development. Neglecting this step results in limiting the system in terms of both scalability and performance, which becomes particularly apparent once the client community grows to unforeseen levels. Once that happens, a latency-blind system is not going to meet client requirements with respect to access latency and throughput, and the constraints of its original design might prevent one from effectively improving the system performance by means of latency-driven techniques.

Latencies in a globally distributed systems can be optimized by means of replication, which reduces the network distance traversed by client requests and system responses. However, from among many available replication techniques, a globally distributed system should select those focused on latency optimization. Replica placement algorithms play the key role here, as replica locations effectively determine the lower bound on network latencies that can be offered to the clients. In particular, latency-aware redirection policies will be of little help when the replica locations have been chosen without considering network latencies.

The second major conclusion is that the limitations of individual nodes located in a given area do not preclude the use of latency-driven techniques, as one can compensate for these limitations by grouping nodes into high-performance distributed servers. Our proposed techniques provide nodes scattered over a wide-area network with network-level distribution transparency, which is implemented without relying on any physical frontends. The distribution transparency enables distributed servers to employ techniques previously available only in local-area networks, such as TCP handoff. More importantly, however, each distributed server can also preserve its network address despite (potentially total) changes in its composition. This feature allows for treating distributed servers as IP-addressable logical entities that are not bound to any physical nodes. We envisage that Internet services can employ our solutions to turn dynamically changing groups of nodes located in different network regions into virtual hosting facilities for service replicas.

We believe that the solutions presented in this thesis can be applied in a great variety of existing Internet services whose performance depends on fast communication between their individual components. We are also convinced that our research will be even more useful to researchers and engineers developing future global systems that can still be designed in a latency-aware manner. We anticipate that our techniques will allow for constructing systems in which fast communication is guaranteed regardless of tremendous system size and global distribution.

SAMENVATTING

Tijdsafhankelijke Replicatie voor Wereldwijd Gespreide Systemen

In de afgelopen twee decennia is het Internet geëvolueerd van een experimenteel onderzoeksnetwerk tot een wereldwijd communicatie medium. Het wordt nu dagelijks gebruikt door miljoenen mensen om email te versturen, nieuws te lezen op het Web of om te winkelen op duizenden *e-commerce* sites.

De groei van het Internet stelt de architecten van Internet diensten voor nieuwe uitdagingen. Om succesvol te zijn dient een Internet dienst makkelijk in gebruik te blijven ondanks voorziene toenames in zowel netwerkgrootte als aantallen gebruikers. Vanuit menselijk oogpunt kan gebruiksgemak slaan op veel verschillende eigenschappen, die feitelijk in twee groepen opgesplitst kunnen worden. De eerste groep bestaat uit de eigenschappen van de informatie die geretourneerd wordt door de dienst, zoals hoe makkelijk het is om door deze informatie te navigeren, of de beschikbaarheid in meerdere talen. Deze eigenschappen worden over het algemeen onderzocht in studies naar mens-computer interactie.

De tweede groep van eigenschappen die gebruiksgemak beïnvloeden bestaat uit die welke gerelateerd zijn aan de kwaliteit van de communicatie met de dienst. Tot deze eigenschappen behoren, bijvoorbeeld, de beschikbaarheid van de dienst, communicatiebeveiliging en responstijd van de dienst op verzoeken van de gebruiker. Terwijl individuele beoordelingen van de kwaliteit van de geretourneerde informatie van nature subjectief zijn, verwachten alle gebruikers over het algemeen dat Internet diensten altijd beschikbaar en veilig te gebruiken zijn en zo snel mogelijk reageren.

De hoge verwachtingen van gebruikers zijn de motivatie voor onderzoek naar het verbeteren van de communicatiekwaliteit van Internet diensten. Hoofdstuk 2 analyseert een groot aantal onderzoeken naar verscheidene aspecten van zulke communicatie. We classificeren en bediscussiëren oplossingen voor elk aspect

apart, en identificeren de onderzoeksgebieden welke in het verleden relatief zelden onderzocht zijn. Dit brengt ons tot de hoofdonderzoeksvraag van dit proefschrift: *hoe kunnen de responstijden van grootschalige Internet diensten verbeterd worden?*

In principe bestaat de responstijd van een Internet dienst uit twee delen. Ten eerste, als de dienst een verzoek ontvangt heeft deze enige tijd nodig om zijn antwoord te *genereren*. Ten tweede, is er nog wat meer tijd nodig om het genereerde antwoord te *versturen* naar de gebruikersmachine. De technieken die in dit proefschrift geïntroduceerd worden hebben tot doel de transmissietijd te reduceren, en zijn onafhankelijk van de technieken voor het genereren van antwoorden. Als gevolg hiervan doen onze technieken alsof de antwoorden van te voren gegenereerd zijn, wat equivalent is aan de aanname dat alle data statisch gerepliceerd is. Merk op dat deze aanname niet uitsluit dat de door ons voorgestelde technieken gebruikt worden in systemen die antwoorden ter plekke genereren. Zulke systemen kunnen namelijk altijd onze oplossingen combineren met hun eigen technieken voor het dynamisch genereren van respons.

Transmissietijden zijn in essentie van drie factoren afhankelijk: De eerste factor is de netwerkcapaciteit beschikbaar op het pad tussen een Internet dienst en een gegeven gebruikersmachine, welke het tempo bepaalt waarin de Internet dienst zijn antwoorden naar die machine zendt. De tweede factor is de vertraging tussen de Internet dienst en de gebruikersmachine, wat eenvoudigweg de benodigde tijd is om de kleinste hoeveelheid data tussen de communicerende partijen te transporteren. De derde factor is de frequentie van pakketverlies, welke definiëert hoeveel data er gemiddeld opnieuw verzonden moet worden om ervoor te zorgen dat alle data succesvol overgedragen wordt.

Een cruciale observatie is dat de bovenstaande drie factoren niet dezelfde relevantie hebben bij het optimaliseren van de transmissietijd. Pakketverlies, bijvoorbeeld, wat typisch voorkomt in overbelaste netwerken dicht bij de gebruikersmachines, kan niet verholpen worden door Internet diensten die geen invloed hebben op zulke netwerken. Zo is ook de doorvoersnelheid van TCP connecties, die vaak voor communicatie met Internet diensten gebruikt worden, gelimiteerd door netwerkvertragingen tussen communicerende partijen. Gegeven dat deze beperking zelfs bestaat in netwerken van oneindige capaciteit is het onwaarschijnlijk dat transmissietijden alleen verbeterd kunnen worden door netwerkcapaciteit te overdimensioneren. In plaats daarvan moet het vergroten van de netwerkcapaciteit gepaard gaan met het optimaliseren van netwerkvertragingen tussen de Internet dienst en de gebruikersmachines. Wij richten ons daarom op het optimaliseren van zulke vertragingen als het sleutelvraagstuk bij het optimaliseren van transmissietijden.

Het fundamentele probleem bij vertragingsoptimalisatie is dat de vertragingen van (niet-overbelaste) netwerkpaden bepaald worden door de fysieke eigenschappen van de verbindingen waaruit deze paden opgebouwd zijn, zoals de lichtsnelheid in een gegeven fysiek medium. Aangezien het veranderen van deze eigenschappen onmogelijk is, is de enige manier om vertragingen te reduceren het reorganiseren van de communicatie met de Internet dienst zodat antwoorden over korte netwerkpaden worden verzonden.

Echter, wanneer de gebruikers verspreid zijn over de gehele wereld is het onmogelijk met elke gebruikersmachine te communiceren via een kort netwerkpad. Grootschalige Internet diensten lossen dit probleem vaak op door middel van replicatie, waarbij meerder instanties van de dienst aangeboden worden op verschillende plaatsen in de wereld. Elke instantie kan dan verzoeken afhandelen van gebruikersmachines in zijn nabijheid, waardoor korte netwerkpaden gebruikt kunnen worden voor communicatie. Dit resulteert in lage netwerkvertragingen tussen de dienst en zijn cliënten, wat de responstijd van de dienst verlaagt en waardoor de dienst aantrekkelijker wordt voor zijn gebruikers.

Replicatie wordt algemeen toegepast door veel verschillende soorten Internet diensten, in het bijzonder diensten die grote datacollecties beheren. Voorbeelden zijn *content-delivery networks*, *peer-to-peer* systemen voor het delen van bestanden, en databanken. Het meeste van het recente onderzoek naar wereldwijde replicatie is echter gedaan in de context van het Web. Wij richten ons daarom ook op de Web omgeving voor onze oplossingen, alhoewel veel van hen ook toepasbaar zijn in andere systemen.

Systemen die Web replicatie implementeren worden gewoonlijk aangeduid als *Web replica hosting systems*. Zo'n systeem bestaat uit een aantal replicaservers, die replica's bevatten van Web documenten die door het systeem aangeboden worden. Wanneer een Web cliënt zo'n document wenst te downloaden dan verwijst het systeem de Web cliënt naar een nabijgelegen replicaserver waar een replica van het document aanwezig is. De Web cliënt kan dan contact opnemen met de replicaserver en de inhoud van het document ophalen.

Hoewel de werking van een *Web replica hosting system* relatief simpel lijkt moet zo'n systeem veel specifieke problemen oplossen. Het behouden van hoge systeemprestaties, bijvoorbeeld, vereist dat deze continu geëvalueerd worden zodat het systeem weet wanneer het actie moet ondernemen. *Vertragingsgestuurde* systemen in het bijzonder moeten in staat zijn om vertragingen tussen hun individuele knopen (cliënt en replicaservers) te schatten. Zulke informatie over vertragingen is noodzakelijk om juiste beslissingen te nemen over het creëren van nieuwe replica's op specifieke replicaservers, en deze te configureren voor gebruik door een gegeven groep van cliënten. In een vertragingsgestuurd system

moeten deze beide beslissingen genomen worden met behulp van technieken die *vertragingsbewust* zijn.

Dit proefschrift presenteert een aantal vertragingsbewuste replicatietechnieken voor grootschalige systemen. We laten zien dat verscheidene problemen bijzonder moeilijk zijn wanneer een systeem bestaat uit miljoenen knopen, en stellen oplossingen voor die specifiek ontworpen zijn voor zulke situaties. Deze oplossingen concentreren zich op drie zaken die naar ons inzicht de transmissietijd het meest beïnvloeden: modellering van vertraging, replicaplaatsing en het omleiden van cliënten.

Het schatten van de vertraging tussen twee knopen in een grootschalig systeem is lastig door het enorme aantal paren knopen. Omdat vertragingen de neiging hebben om continu te fluctueren is het herhaaldelijk meten van vertragingen tussen alle paren van knopen niet praktisch haalbaar. In plaats daarvan moet het systeem technieken toepassen die grote aantallen vertragingsschattingen opleveren tegen lage kosten. Onze oplossingen benutten het concept van netwerkpositionering, wat het Internet modelleert als een multidimensionale geometrische ruimte. De lage kosten voor de modellering en het feit dat het model geometrisch is maken dit concept bijzonder aantrekkelijk voor grootschalige systemen. Hoofdstuk 3 bediscussiëert hoe netwerkpositionering efficiënt geïmplementeerd kan worden in zowel gecentraliseerde als gefedereerde omgevingen, en presenteert verscheidene experimenten gebaseerd op echte metingen om de hoge accuratesse aan te tonen van de verkregen vertragingsschattingen.

Systeemmodellen gebaseerd op netwerkpositionering kunnen gebruikt worden om te besluiten waar replica's geplaatst moeten worden. Het probleem is echter dat replica's over het algemeen gebruikt worden door cliënten uit meerdere delen van het Internet. Het identificeren van replica-locaties die netwerkvertragingen voor alle cliënten minimaliseren is niet triviaal, en vereisten tot nu toe tijdrovende berekeningen waarbij gigantische aantallen paarsgewijze vertragingen gemeten werden. Zulke berekeningen zijn niet praktisch haalbaar in een echt systeem dat beslist over de plaatsing van vele replica's. Wij stellen dan ook een nieuw algoritme voor de plaatsing van replica's voor dat de geometrische eigenschappen exploiteert van modellen opgesteld met behulp van netwerkpositionering. Het identificeert de beste replica-locaties als de clusters van knooppositities in de geometrische ruimte, en beeldt replica's zorgvuldig af op clusters om overlap te voorkomen. Hoofdstuk 4 beschrijft de details van ons algoritme, en toont aan dat het replica posities van hoge kwaliteit genereert en wel verscheidene ordes van grootte sneller dan voorgaande oplossingen.

De clusters die gekozen worden door ons replica-plaatsingsalgoritme corresponderen met netwerkregio's in plaats van individuele machines. Als gevolg hiervan moet een andere verzameling technieken gebruikt worden om de replica's

af te beelden op individuele replicaservers in deze regio's. Omdat de eigenschappen van onze vertragingsmodellen echter garanderen dat alle replicaservers in een gegeven regio vergelijkbare vertraging hebben, kunnen deze technieken zich richten op het optimaliseren van andere metrieken. Wij hebben netwerkbandbreedte en de beschikbaarheid van servers als metrieken gekozen en stellen voor deze te verbeteren door de replicaservers in een regio te organiseren als een gespreide server. Zo'n server kan meer netwerkbandbreedte bieden door de capaciteiten van de netwerkverbindingen van de individuele replicaservers te aggregeren, en is in staat een stabiel netwerkadres te behouden ondanks eventueel falen van individuele replicaservers. Dit stelt elke gespreide server in staat om een zichzelf organiserend platform te worden dat een groep potentieel onbetrouwbare replicaservers tot een betrouwbare exploitatie-faciliteit voor Internet diensten maakt. Hoofdstuk 5 presenteert onze technieken voor het realiseren van stabiele netwerkadressen voor gespreide servers, en aggregatie van bandbreedte door middel van transparante, wereldwijde overdracht van cliënten tussen servers.

Hoewel het intraregionaal omleiden van cliënten afgehandeld wordt door de gespreide servers, zijn voor zijn interregionale tegenhanger andere technieken nodig. Eén van de problemen die deze technieken moeten oplossen is het kiezen van de gespreide server voor elke cliënt zodanig dat de paarsgewijze vertraging tussen de twee geminimaliseerd wordt. Ons voorstel is om dit probleem op te lossen met onze vertragingsmodellen, die ons in staat stellen om de vertragingen tussen cliënt en alle gespreide servers te schatten, en daarna de beste kandidaatserver te identificeren. Een ander probleem is hoe de cliënt op de hoogte wordt gesteld van de beslissing tot omleiden. Dit kan gedaan worden met behulp van DNS, wat hiervoor algemeen gebruikt wordt door grootschalige systemen. Aan het einde van Hoofdstuk 5, bediscussiëren we hoe wereldwijde omleiding van cliënten geïmplementeerd kan worden door het combineren van wereldwijde overdrachten met mechanismen voor domeinnaam-resolutie.

We zijn van mening dat onze voorgestelde verstragingsgestuurde technieken toepasbaar zijn in een verscheidenheid aan grootschalige gespreide systemen. Ondanks dat ze in de context van het Web beschreven zijn, zijn alle technieken generiek genoeg om ook toegepast te worden in andere systemen die de netwerkvertragingen tussen hun componenten willen optimaliseren. Hoofdstuk 6 legt uit hoe ieder van onze technieken gezien kan worden als een stap in de richting van vertragingsminimalisatie in een wereldwijd gespreid system, en stelt een aantal richtingen voor toekomstig onderzoek voor voortbouwend op onze resultaten.

BIBLIOGRAPHY

ABOBA, B., ARKKO, J., AND HARRINGTON, D. 2000. Introduction to Accounting Management. RFC 2975.

ACHARYA, A. AND SHAIKH, A. 2002. Using Mobility Support for Request-Routing in IPv6 CDNs. In *Proc. 7th International Workshop on Web Content Caching and Distribution*.

AFERGAN, M., WEIN, J., AND LAMEYER, A. 2005. Experience with some Principles for Building an Internet-Scale Reliable System. In *Proc. 2nd Workshop on Real Large Distributed Systems*. USENIX, Berkeley, CA, 1–6.

AGGARWAL, A. AND RABINOVICH, M. 1998. Performance of Replication Schemes for an Internet Hosting Service. Tech. Rep. HA6177000-981030- 01-TM, AT&T Research Labs, Florham Park, NJ. Oct.

ALLMAN, M. 2000. A Web server's View of the Transport Layer. *ACM Computer Communications Review 30*, 5, 10–20.

ALMESBERGER, W. 2004. TCP Connection Passing. In *Proc. Ottawa Linux Symposium*. [Online] http://www.linuxsymposium.org/.

ALVISI, L. AND MARZULLO, K. 1998. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering 24*, 2 (Feb.), 149–159.

AMINI, L. AND SCHULZRINNE, H. 2004. Client Clustering for Traffic and Location Estimation. In *Proc. 24th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA., 730–737.

AMINI, L., SHAIKH, A., AND SCHULZRINNE, H. 2003. Modeling Redirection in Geographically Diverse Server Sets. In *Proc. 12th International World Wide Web Conference*. ACM Press, New York, NY, 472–481.

AMINI, L., SHAIKH, A., AND SCHULZRINNE, H. 2004. Effective Peering for Multi-provider Content Delivery Services. In *Proc. 23rd INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 850–861.

AMIRI, K., PARK, S., TEWARI, R., AND PADMANABHAN, S. 2003. DBProxy: A Dynamic Data Cache for Web Applications. In *Proc. 19th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA., 821–831.

ANDREOLINI, M., COLAJANNI, M., AND NUCCIO, M. 2003a. Kernel-based Web Switches Providing Content-Aware Routing. In *Proc. 2nd International Symposium on Network Computing and Applications*. IEEE Computer Society Press, Los Alamitos, CA., 25–32.

ANDREOLINI, M., COLAJANNI, M., AND NUCCIO, M. 2003b. Scalability of Content-Aware Server Switches for Cluster-Based Web Information Systems. In *Proc. 12th International World Wide Web Conference*. ACM Press, New York, NY.

ANDREWS, M., SHEPHERD, B., SRINIVASAN, A., WINKLER, P., AND ZANE, F. 2002. Clustering and Server Selection Using Passive Monitoring. In *Proc. 21st INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1717–1725.

ARDAIZ, O., FREITAG, F., AND NAVARRO, L. 2001. Improving the Service Time of Web Clients using Server Redirection. In *Proc. 2nd Workshop on Performance and Architecture of Web Servers*. ACM Press, New York, NY, 39–44.

ARKKO, J., DEVARAPALLI, V., AND DUPONT, F. 2004. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. RFC 3776.

BALAKRISHNAN, H., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2003. Looking up Data in P2P Systems. *Communications of the ACM 46,* 2 (Feb.), 43–48.

BALLINTIJN, G., VAN STEEN, M., AND TANENBAUM, A. 2000. Characterizing Internet Performance to Support Wide-area Application Development. *Operating Systems Review 34,* 4 (Oct.), 41–47.

BARBIR, A., CAIN, B., DOUGLIS, F., GREEN, M., HOFFMAN, M., NAIR, R., POTTER, D., AND SPATSCHECK, O. 2002. Known CDN Request-Routing Mechanisms. Internet Draft.

BARFORD, P., CAI, J.-Y., AND GAST, J. 2001. Cache Placement Methods Based on Client Demand Clustering. Tech. Rep. TR1437, University of Wisconsin at Madison.

BARROSO, L., DEAM, J., AND HÖLZE, U. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro 23,* 2 (Mar.), 21–28.

BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., , AND WAWRZONIAK, M. 2004. Operating System Support for Planetary-Scale Network Services. In *Proc. Symposium on Networked Systems Design and Implementation*. USENIX, Berkeley, CA, 253–266.

BERNSTEIN, P. A. AND GOODMAN, N. 1983. The Failure and Recovery Problem for Replicated Databases. In *Proc. 2nd Symposium on Principles of Distributed Computing*. ACM Press, New York, NY, 114–122.

BHIDE, M., DEOLASEE, P., KATKAR, A., PANCHBUDHE, A., RAMAMRITHAM, K., AND SHENOY, P. 2002. Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers 51,* 6 (June), 652–668.

BORNHOVD, C., ALTINEL, M., MOHAN, C., PIRAHESH, H., AND REINWALD, B. 2004. Adaptive Database Caching with DBCache. *Data Engineering 27,* 11–18.

BREWER, E. 2001. Lessons from Giant-Scale Services. *IEEE Internet Computing 5,* 4, 46–55.

BRIN, S. AND PAGE, L. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems 30,* 1–7, 107–117.

BRISCO, T. 1995. DNS Support for Load Balancing. RFC1794.

BUCHSBAUM, A. L., FOWLER, G. S., KIRISHNAMURTHY, B., VO, K.-P., AND WANG, J. 2003. Fast Prefix Matching of Bounded Strings. *J. Exp. Algorithmics 8,* 1.3.

CAO, J., DAVIS, D., WIEL, S. V., AND YU, B. 2000. Time-Varying Network Tomography: Router Link Data. In *Proc. International Symposium on Information Theory*. IEEE Computer Society Press, Los Alamitos, CA., 79–90.

CAO, P. AND LIU, C. 1998. Maintaining Strong Cache Consistency in the World Wide Web. *IEEE Transactions on Computers 47,* 4 (Apr.), 445–457.

CARDELLINI, V., CASALICCHIO, E., COLAJANNI, M., AND YU, P. 2002. The State of the Art in Locally Distributed Web-Server Systems. *ACM Computing Surveys 34*, 2 (June), 263–311.

CARDELLINI, V., COLAJANNI, M., AND YU, P. 1999. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing 3,* 3 (May), 28–39.

CARDELLINI, V., COLAJANNI, M., AND YU, P. S. 2003. Request Redirection Algorithms for Distributed Web Systems. *IEEE Transactions on Parallel and Distributed Systems 14,* 4 (Apr.), 355–368.

CARSON, M. AND SANTAY, D. 2003. NIST Net: a Linux-based Network Emulation Tool. *ACM Computer Communications Review 33*, 3, 111–126.

CARTER, R. L. AND CROVELLA, M. E. 1997. Dynamic Server Selection Using Bandwidth Probing in Wide-Area Networks. In *Proc. 16th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1014–1021.

CASTRO, M., COSTA, M., ROWSTRON, A., AND KEY, P. 2004. PIC: Practical Internet Coordinates for Distance Estimation. In *Proc. 24th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA., 178–187.

CASTRO, M., DRUSCHEL, P., HU, Y. C., AND ROWSTRON, A. 2003. Proximity Neighbor Selection in Tree-based Structured Peer-to-Peer Overlays. Tech. Rep. MSR-TR-2003-52, Microsoft Research, Cambridge, UK.

CASTRO, M., DRUSCHEL, P., KERMARREC, A., AND ROWSTRON, A. 2003. Scalable Application-Level Anycast for Highly Dynamic Groups. In *Proc. Networked Group Communication Workshop*. Lecture Notes on Computer Science, vol. 2816. Springer-Verlag, Berlin, 47–57.

CATE, V. 1992. Alex – A Global File System. In *Proc. File Systems Workshop*. USENIX, Berkeley, CA, 1–11.

CHANDRA, P., CHU, Y.-H., FISHER, A., GAO, J., KOSAK, C., NG, T. E., STEENKISTE, P., TAKAHASHI, E., AND ZHANG, H. 2001. Darwin: Customizable Resource Management for Value-Added Network Services. *IEEE Network 1,* 15 (Jan.), 22–35.

CHEN, Y., KATZ, R., AND KUBIATOWICZ, J. 2002. Dynamic Replica Placement for Scalable Content Delivery. In *Proc. 1st International Workshop on Peer-to-Peer Systems*. Lecture Notes on Computer Science, vol. 2429. Springer-Verlag, Berlin, 306–318.

CHEN, Y., QIU, L., CHEN, W., NGUYEN, L., AND KATZ, R. H. 2002. Clustering Web Content for Efficient Replication. In *Proc. 10th International Conference on Network Protocols*. IEEE Computer Society Press, Los Alamitos, CA., 165–174.

CHEN, Y., QIU, L., CHEN, W., NGUYEN, L., AND KATZ, R. H. 2003. Efficient and Adaptive Web Replication Using Content Clustering. *IEEE Journal on Selected Areas in Communication 21,* 6 (Aug.), 979–994.

CHUN, B., WU, P., WEATHERSPOON, H., AND KUBIATOWICZ, J. 2006. ChunkCast: An Anycast Service for Large Content Distribution. In *Proc. 5th International Workshop on Peer-to-Peer Systems*.

COATES, M., CASTRO, R., AND NOWAK, R. 2002. Maximum Likelihood Network Topology Identification from Edge-based Unicast Measurements. In *Proc. International Conference on Measurements and Modeling of Computer Systems*. ACM Press, New York, NY, 11–20.

COHEN, B. 2006. Official BitTorrent Home Page. [Online] http://www.bittorrent.com/.

COHEN, E. AND KAPLAN, H. 2001. Proactive Caching of DNS Records: Addressing a Performance Bottleneck. In *Proc. 1st Symposium on Applications and the Internet*. IEEE Computer Society Press, Los Alamitos, CA., 85–94.

COHEN, E. AND SHENKER, S. 2002. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proc. SIGCOMM*. ACM Press, New York, NY, 177–190.

CONTI, M., GREGORI, E., AND LAPENNA, W. 2002. Replicated Web Services: A Comparative Analysis of Client-Based Content Delivery Policies. In *Proc. Networking 2002 Workshops*. Lecture Notes on Computer Science, vol. 2376. Springer-Verlag, Berlin, 53–68.

COX, R., DABEK, F., KAASHOEK, F., LI, J., AND MORRIS, R. 2004. Practical, Distributed Network Coordinates. *ACM Computer Communications Review 34,* 1 (Jan.), 113–118.

CROVELLA, M. AND CARTER, R. 1995. Dynamic Server Selection in the Internet. In *Proc. 3rd Workshop on High Performance Subsystems*. IEEE Computer Society Press, Los Alamitos, CA., 158–162.

DA CUNHA, C. R. 1997. Trace Analysis and its Applications to Performance Enhancements of Distributed Information Systems. Ph.D. Thesis, Boston University.

DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. 2004. Vivaldi: A Decentralized Network Coordinate System. In *Proc. SIGCOMM*. ACM Press, New York, NY, 15–26.

DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M., AND MORRIS, R. 2004. Designing a DHT for Low Latency and High Throughput. In *Proc. Symposium on Networked Systems Design and Implementation*. USENIX, Berkeley, CA.

DAVIS, A., PARIKH, J., AND WEIHL, W. E. 2004. EdgeComputing: Extending Enterprise Applications to the Edge of the Internet. In *Proc. 13th International World Wide Web Conference*. ACM Press, New York, NY, 180–187.

DELGADILLO, K. 1999. Cisco DistributedDirector. Tech. rep., Cisco Systems, Inc. June.

DILLEY, J., MAGGS, B., PARIKH, J., PROKOP, H., SITARAMAN, R., AND WEIHL, B. 2002. Globally Distributed Content Delivery. *IEEE Internet Computing 6,* 5 (Sept.), 50–58.

DUCHAMP, D. 1999. Prefetching Hyperlinks. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems*. USENIX, Berkeley, CA.

DUFFIELD, N. 2003. Simple Network Performance Tomography. In *Proc. ACM Internet Measurement Conference*. ACM Press, New York, NY, 210–215.

DUFFIELD, N., PRESTI, F. L., PAXSON, V., AND TOWSLEY, D. 2001. Inferring Link Loss Using Striped Unicast Probes. In *Proc. 20th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 915–923.

DUVVURI, V., SHENOY, P., AND TEWARI, R. 2000. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proc. 19th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 834–843.

DYKES, S. G., ROBBINS, K. A., AND JEFFREY, C. L. 2000. An Empirical Evaluation of Client-side Server Selection. In *Proc. 19th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1361–1370.

EDWARDS, A. L. 1976. The Correlation Coefficient. In *An Introduction to Linear Regression and Correlation*. W. H. Freeman, San Francisco, CA, Chapter 4, 33–46.

ELNOZAHY, E., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. 2002. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys 34,* 3 (Sept.), 375–408.

FEI, Z. 2001. A Novel Approach to Managing Consistency in Content Distribution Networks. In *Proc. 6th International Workshop on Web Content Caching and Distribution*.

FEI, Z., BHATTACHARJEE, S., ZEGURA, E. W., AND AMMAR, M. 1998. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proc. 17th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 783–791.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.

FIREFOX. 2006. Official Firefox Home Page. [Online] http://www.mozilla.com/firefox/.

FISHER, D. AND SAKSENA, G. 2003. SYNOPSIS: Link Prefetching in Mozilla: A Server-Driven Approach. In *Proc. 8th International Workshop on Web Content Caching and Distribution*.

FOSTER, I. AND IAMNITCHI, A. 2003. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proc. 2nd International Workshop on Peer-to-Peer Systems*. Lecture Notes on Computer Science, vol. 2735. Springer-Verlag, Berlin, 118–128.

FRALEIGH, C., TOBAGI, F., AND DIOT, C. 2003. Provisioning IP Backbone Networks to Support Latency Sensitive Traffic. In *Proc. 22nd INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 375–385.

FRANCIS, P., JAMIN, S., JIN, C., JIN, Y., RAZ, D., SHAVITT, Y., AND ZHANG, L. 2001. IDMaps: Global Internet Host Distance Estimation Service. *IEEE/ACM Transactions on Networking 9*, 5 (Oct.), 525–540.

FRANCIS, P., JAMIN, S., PAXSON, V., ZHANG, L., GRYNIEWICZ, D., AND JIN, Y. 1999. An Architecture for a Global Internet Host Distance Estimation Service. In *Proc. 18th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 210–217.

FREEDMAN, M. J., LAKSHMINARAYANAN, K., AND MAZIERES, D. 2006. OASIS: Anycast for Any Service. In *Proc. 3rd Symposium on Networked Systems Design and Implementation*. USENIX, Berkeley, CA, 129–142.

FU, Y., CHERKASOVA, L., TANG, W., AND VAHDAT, A. 2002. EtE: Passive End-to-End Internet Service Performance Monitoring. In *Proc. USENIX Annual Technical Conference*. USENIX, Berkeley, CA, 115–130.

GANGULY, A., AGRAWAL, A., BOYKIN, P., AND FIGUEIREDO, R. 2006. WOW: Self-organizing Wide Area Overlay Networks of Virtual Workstations. In *Proc. 15th International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos, CA., 30–42.

GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., AND IYENGAR, A. 2003. Application Specific Data Replication for Edge Services. In *Proc. 12th International World Wide Web Conference*. ACM Press, New York, NY, 449–460.

GEORGATOS, F., GRUBER, F., KARRENBERG, D., SANTCROOS, M., SUSANJ, A., UIJTERWAAL, H., AND WILHELM, R. 2001. Providing Active Measurements as a Regular Service for ISP's. In *Proc. Passive and Active Measurement Workshop*.

GRAY, C. AND CHERITON, D. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. 12th Symposium on Operating System Principles*. ACM Press, New York, NY, 202–210.

GUMMADI, K. P., SAROIU, S., AND GRIBBLE, S. D. 2002. King: Estimating Latency between Arbitrary Internet End Hosts. In *Proc. 2nd Internet Measurement Workshop*. ACM Press, New York, NY, 5–18.

HABERMAN, B. AND NORDMARK, E. 2002. IPv6 Anycast Binding using Return Routability. Internet Draft.

HARKINS, D. AND CARREL, D. 1998. The Internet Key Exchange (IKE). RFC 2409.

HUFFAKER, B., FOMENKOV, M., PLUMMER, D. J., MOORE, D., AND CLAFFY, K. 2002. Distance Metrics in the Internet. In *Proc. International Telecommunications Symposium*. IEEE Computer Society Press, Los Alamitos, CA.

HULL, S. 2002. *Content Delivery Networks*. McGraw-Hill, New York, NY.

HUSTON, G. 2004. Anatomy: A Look Inside Network Address Translators. *Internet Protocol Journal 7,* 3 (Sept.).

IANNACCONE, G., CHUAH, C., MORTIER, R., BHATTACHARYYA, S., AND DIOT, C. 2002. Analysis of Link Failures in an IP Backbone. In *Proc. 2nd Internet Measurement Workshop*. ACM Press, New York, NY, 237–242.

JAIN, M. AND DOVROLIS, C. 2002. Pathload: a Measurement Tool for End-to-End Available Bandwidth. In *Proc. Passive and Active Measurement Workshop*.

JALOTE, P. 1994. *Fault Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliffs, N.J.

JAMIN, S., JIN, C., KURC, A. R., RAZ, D., AND SHAVITT, Y. 2001. Constrained Mirror Placement on the Internet. In *Proc. 20th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 31–40.

JANIGA, M. J., DIBNER, G., AND GOVERNALI, F. J. 2001. Internet Infrastructure: Content Delivery. Goldman Sachs Global Equity Research.

JOHNSON, D., PERKINS, C., AND ARKKO, J. 2004. Mobility Support in IPv6. RFC 3775.

JOHNSON, K. L., CARR, J. F., DAY, M. S., AND KAASHOEK, M. F. 2001. The Measured Performance of Content Distribution Networks. *Computer Communications 24,* 2 (Feb.), 202–206.

JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. 2002. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proc. 11th International World Wide Web Conference*. ACM Press, New York, NY, 293–304.

JUNG, J., SIT, E., BALAKRISHNAN, H., AND MORRIS, R. 2002. DNS Performance and the Effectiveness of Caching. *IEEE/ACM Transactions on Networking 10,* 5 (Oct.), 589 – 603.

KANGASHARJU, J., ROBERTS, J., AND ROSS, K. 2001. Object Replication Strategies in Content Distribution Networks. In *Proc. 6th International Workshop on Web Content Caching and Distribution*.

KANGASHARJU, J., ROSS, K., AND ROBERTS, J. 2001. Performance Evaluation of Redirection Schemes in Content Distribution Networks. *Computer Communications 24,* 2 (Feb.), 207–214.

KARAUL, M., KORILIS, Y., AND ORDA, A. 1998. A Market-Based Architecture for Management of Geographically Dispersed, Replicated Web Servers. In *Proc. 1st International Conference on Information and Computation Economics*. ACM Press, New York, NY, 158–165.

KARGER, D., SHERMAN, A., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., AND YERUSHALMI, Y. 1999. Web Caching with Consistent Hashing. In *Proc. 8th International World Wide Web Conference*. ACM Press, New York, NY, 1203–1213.

KARLSSON, M. AND KARAMANOLIS, C. 2004. Choosing Replica Placement Heuristics for Wide-Area Systems. In *Proc. 24th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA., 350–359.

KARLSSON, M., KARAMANOLIS, C., AND MAHALINGAM, M. 2002. A Framework for Evaluating Replica Placement Algorithms. Tech. Rep. HPL-2002-219, HP Laboratories, Palo Alto, CA. aug.

KARLSSON, M. AND MAHALINGAM, M. 2002. Do We Need Replica Placement Algorithms in Content Delivery Networks? In *Proc. 7th International Workshop on Web Content Caching and Distribution*. 117–128.

KARVE, A., KIMBREL, T., PACIFICI, G., SPREITZER, M., STEINDER, M., SVIRIDENKO, M., AND TANTAWI, A. 2006. Dynamic Placement for Clustered Web Applications. In *Proc. 15th International World Wide Web Conference*. ACM Press, New York, NY, 595–604.

KOKKU, R., YALAGANDULA, P., VENKATARAMANI, A., AND DAHLIN, M. 2003. NPS: A Non-interfering Deployable Web Prefetching System. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*. USENIX, Berkeley, CA.

KRISHNAKUMAR, N. AND BERNSTEIN, A. J. 1994. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems 4*, 19, 586–625.

KRISHNAMURTHY, B. AND WANG, J. 2000. On Network-Aware Clustering of Web Clients. In *Proc. SIGCOMM*. ACM Press, New York, NY, 97–110.

KUDALLUR, V. 2005. IE7 Networking Improvements in Content Caching and Decompression. [Online] http://blogs.msdn.com/.

LAI, K. AND BAKER, M. 1999. Measuring Bandwidth. In *Proc. 18th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 235–245.

LEE, S., ZHANG, Z., SAHU, S., AND SAHA, D. 2006. On Suitability of Euclidean Embedding of Internet Hosts. In *Proc. International Conference on Measurements and Modeling of Computer Systems*. ACM Press, New York, NY, 157–168.

LEIGHTON, F. AND LEWIN, D. 2000. Global Hosting System. United States Patent, Number 6,108,703.

LI, B., GOLIN, M. J., ITALIANO, G. F., AND DENG, X. 1999. On the Optimal Placement of Web Proxies in the Internet. In *Proc. 18th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1282–1290.

LIM, H., HOU, J., AND CHOI, C.-H. 2003. Constructing Internet Coordinate System Based on Delay Measurement. In *Proc. ACM Internet Measurement Conference*. ACM Press, New York, NY, 129–142.

LUA, E. K., GRIFFIN, T., PIAS, M., ZHENG, H., AND CROWCROFT, J. 2005. On the Accuracy of Embeddings for Internet Coordinate Systems. In *Proc. ACM Internet Measurement Conference*. USENIX, Berkeley, CA, 125–138.

MAO, Z., REXFORD, J., WANG, J., AND KATZ, R. 2003. Towards an Accurate AS-Level Traceroute Tool. In *Proc. SIGCOMM*. ACM Press, New York, NY, 365–378.

MAO, Z. M., CRANOR, C. D., DOUGLIS, F., RABINOVICH, M., SPATSCHECK, O., AND WANG, J. 2002. A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers. In *Proc. USENIX Annual Technical Conference*. USENIX, Berkeley, CA, 229–242.

MCCUNE, T. AND ANDRESEN, D. 1998. Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters. In *Proc. 7th International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos, CA., 301–309.

MCMANUS, P. R. 1999. A Passive System for Server Selection within Mirrored Resource Environments Using AS Path Length Heuristics. Tech. rep., AppliedTheory Corp.

MEDINA, A., TAFT, N., SALAMATIAN, K., BHATTACHARYYA, S., AND DIOT, C. 2002. Traffic Matrix Estimation: Exisiting Techniques and New Directions. In *Proc. SIGCOMM*. ACM Press, New York, NY, 161–174.

MIPL. 2006. Mobile IPv6 for Linux. [Online] http://www.mobile-ipv6.org/.

MIPV6-SRL. 2006. MIPv6 Systems Research Lab. [Online] http://www.mobileipv6.net/.

MOCKAPETRIS, P. 1987a. Domain Names - Concepts and Facilities. RFC 1034.

MOCKAPETRIS, P. 1987b. Domain Names - Implementation and Specification. RFC 1035.

MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. 1997. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proc. SIGCOMM*. ACM Press, New York, NY, 181–194.

MOORE, K. AND SWANY, M. 1999. Sonar - A Network Proximity Service. Tech. Rep. UT-CS-99-429, University of Tennessee.

MOSBERGER, D. 1993. Memory Consistency Models. *Operating Systems Review 27*, 1 (Jan.), 18–26.

NELDER, J. A. AND MEAD, R. 1965. A Simplex Method for Function Minimization. *The Computer Journal 4*, 7, 303–318.

NG, E. AND ZHANG, H. 2002. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. 21st INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 170–179.

NG, T. S. E. AND ZHANG, H. 2004. A Network Positioning System for the Internet. In *Proc. USENIX Annual Technical Conference*. USENIX, Berkeley, CA, 141–154.

NIELSEN, J. 1999. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Thousand Oaks, CA, USA.

NINAN, A., KULKARNI, P., SHENOY, P., RAMAMRITHAM, K., AND TEWARI, R. 2002. Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *Proc. 11th International World Wide Web Conference*. ACM Press, New York, NY, 1–12.

OBRACZKA, K. AND SILVA, F. 2000. Network Latency Metrics for Server Proximity. In *Proc. Global Telecommunications Conference*. IEEE Computer Society Press, Los Alamitos, CA., 421–427.

ODLYZKO, A. 2001. Internet Pricing and the History of Communications. *Computer Networks 36*, 493–517.

ONESTAT.COM. 2005. Mozilla's Browsers Global Usage Share is Still Growing. [Online] http://www.onestat.com/.

PAI, V., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., AND NAHUM, E. 1998. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 205–216.

PANSIOT, J. AND GRAD, D. 1998. On Routes and Multicast Trees in the Internet. *ACM Computer Communications Review 28*, 1, 41–50.

PARTRIDGE, C., MENDEZ, T., AND MILLIKEN, W. 1993. Host Anycasting Service. RFC 1546.

PAXSON, V. 1997a. End-to-end Routing Behavior in the Internet. *IEEE/ACM Transactions on Networking 5*, 5 (Oct.), 601–615.

PAXSON, V. 1997b. Measurements and Analysis of End-to-End Internet Dynamics. Ph.D. thesis, University of California at Berkeley.

PEPELNJAK, I. AND GUICHARD, J. 2001. *MPLS and VPN Architectures*. Cisco Press, Indianapolis, IN.

PEREIRA, J., RODRIGUES, L., PINTO, A., AND OLIVEIRA, R. 2004. Low Latency Probabilistic Broadcast in Wide Area Networks. In *Proc. 23rd Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA., 299–308.

PETERSEN, K., SPREITZER, M., TERRY, D., AND THEIMER, M. 1996. Bayou: Replicated Database Services for World-wide Applications. In *Proc. 7th SIGOPS European Workshop*. Connemara, Ireland, 275–280.

PIAS, M., CROWCROFT, J., WILBUR, S., HARRIS, T., AND BHATTI, S. 2003. Lighthouses for Scalable Distributed Location. In *Proc. 2nd International Workshop on Peer-to-Peer Systems*. Vol. 2735. Springer-Verlag, Berlin, 278–291.

PIERRE, G., KUZ, I., VAN STEEN, M., AND TANENBAUM, A. 2001. Differentiated Strategies for Replicating Web Documents. *Computer Communications 24*, 2 (Feb.), 232–240.

PIERRE, G. AND VAN STEEN, M. 2006. Globule: a Collaborative Content Delivery Network. *IEEE Communications Magazine 44*, 8 (Aug.), 127–133.

PIERRE, G., VAN STEEN, M., AND TANENBAUM, A. 2002. Dynamically Selecting Optimal Distribution Strategies for Web Documents. *IEEE Transactions on Computers 51*, 6 (June), 637–651.

PIETZUCH, P., LEDLIE, J., AND SELTZER, M. 2005. Supporting Network Coordinates on PlanetLab. In *Proc. 2nd Workshop on Real Large Distributed Systems*. USENIX, Berkeley, CA.

PRADHAN, D. 1996. *Fault-Tolerant Computer System Design*. Prentice Hall, Englewood Cliffs, N.J.

QIU, L., PADMANABHAN, V., AND VOELKER, G. 2001. On the Placement of Web Server Replicas. In *Proc. 20th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1587–1596.

RABINOVICH, M. AND AGGARWAL, A. 1999. Radar: A Scalable Architecture for a Global Web Hosting Service. *Computer Networks 31*, 11–16, 1545–1561.

RABINOVICH, M. AND SPATSCHECK, O. 2002. *Web Caching and Replication*. Addison-Wesley, Reading, MA.

RABINOVICH, M., TRIUKOSE, S., WEN, Z., AND WANG, L. 2006. DipZoom : The Internet Measurements Marketplace. In *Proc. Global Internet Symposium*. IEEE Computer Society Press, Los Alamitos, CA.

RADOSLAVOV, P., GOVINDAN, R., AND ESTRIN, D. 2001. Topology-Informed Internet Replica Placement. In *Proc. 6th International Workshop on Web Content Caching and Distribution*.

RADWARE. 2002. Web Server Director. Tech. rep., Radware, Inc.

RAMABHADRAN, S. AND PASQUALE, J. 2006. Analysis of Long-Running Replicated Systems. In *Proc. 25th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA.

RANJAN, S., KARRER, R., AND KNIGHTLY, E. 2004. Wide Area Redirection of Dynamic Content by Internet Data Centers. In *Proc. 23rd INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 816–826.

RAYNAL, M. AND SINGHAL, M. 1996. Logical Time: Capturing Causality in Distributed Systems. *Computer 29*, 2 (Feb.), 49–56.

REKHTER, Y. AND LI, T. 1995. A Border Gateway Protocol 4 (BGP-4). RFC 1771.

RIBEIRO, V., RIEDI, R., BARANIUK, R., NAVRATIL, J., AND COTTREL, L. 2003. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Proc. Passive and Active Measurement Workshop*.

RODRIGUEZ, P., KIRPAL, A., AND BIERSACK, E. 2000. Parallel-Access for Mirror Sites in the Internet. In *Proc. 19th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 864–873.

RODRIGUEZ, P. AND SIBAL, S. 2000. SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution. *Computer Networks 33*, 1–6, 33–46.

RODRIGUEZ, P., SPANNER, C., AND BIERSACK, E. 2001. Analysis of Web Caching Architecture: Hierarchical and Distributed Caching. *IEEE/ACM Transactions on Networking 21*, 4 (Aug.), 404–418.

SARAT, S., PAPPAS, V., AND TERZIS, A. 2005. On the Use of Anycast in DNS. In *Proc. International Conference on Measurements and Modeling of Computer Systems*. ACM Press, New York, NY, 394–395.

SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. 2002. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. Multimedia Computing and Networking*. SPIE, Bellingham, WA.

SAYAL, M., SHEUERMANN, P., AND VINGRALEK, R. 2003. Content Replication in Web++. In *Proc. 2nd International Symposium on Network Computing and Applications*. IEEE Computer Society Press, Los Alamitos, CA., 33–40.

SCHNEIDER, F. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys 22*, 4 (Dec.), 299–320.

SHAIKH, A., TEWARI, R., AND AGRAWAL, M. 2001. On the Effectiveness of DNS-based Server Selection. In *Proc. 20th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1801–1810.

SHAVITT, Y. AND TANKEL, T. 2004. Big-Bang Simulation for Embedding Network Distances in Euclidean Space. *IEEE/ACM Transactions on Networking 12*, 993–1006.

SHMOYS, D., TARDOS, E., AND AARDAL, K. 1997. Approximation Algorithms for Facility Location Problems. In *Proc. 29th Symposium on Theory of Computing*. ACM Press, New York, NY, 265–274.

SHNEIDERMAN, B. 1984. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys 16*, 3, 265–285.

SHNEIDERMAN, B. 2004. *Designing the User Interface*. Addison Wesley.

SHRIRAM, A., MURRAY, M., HYUN, Y., BROWNLEE, N., BROIDO, A., FOMENKOV, M., AND KC CLAFFY. 2005. Comparison of Public End-to-End Bandwidth Estimation Tools on High-Speed Links. In *Proc. Passive and Active Measurement Workshop*. Lecture Notes on Computer Science, vol. 3431. Springer-Verlag, Berlin, 306–320.

SITEMETER.COM. 2006. SiteMeter Home Page. [Online] http://www.sitemeter.com/.

SIVASUBRAMANIAN, S. 2007. Autonomic Replication for Multitier Web Applications. Ph.D. thesis, Vrije Universiteit Amsterdam, The Netherlands.

SIVASUBRAMANIAN, S., ALONSO, G., PIERRE, G., AND VAN STEEN, M. 2005. GlobeDB: Autonomic Data Replication for Web Applications. In *Proc. 14th International World Wide Web Conference*. ACM Press, New York, NY, 33–42.

SIVASUBRAMANIAN, S., PIERRE, G., AND VAN STEEN, M. 2003. A Case for Dynamic Selection of Replication and Caching Strategies. In *Proc. 8th International Workshop on Web Content Caching and Distribution*.

SIVASUBRAMANIAN, S., PIERRE, G., VAN STEEN, M., AND ALONSO, G. 2006. GlobeCBC: Content-blind Result Caching for Dynamic Web Applications. Tech. Rep. IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands.

SIVASUBRAMANIAN, S., SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. 2004. Replication for Web Hosting Systems. *ACM Computing Surveys 36*, 3, 291–334.

SRINIVASAN, S. AND ZEGURA, E. W. 2004. An Empirical Evaluation of Landmark Placement on Internet Coordinate Schemes. In *Proc. International Conference on Computer Communications and Networks*. IEEE Computer Society Press, Los Alamitos, CA., 335–340.

STEMM, M., KATZ, R., AND SESHAN, S. 2000. A Network Measurement Architecture for Adaptive Applications. In *Proc. 19th INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 285–294.

STRIBLING, J. 2006. All-pairs Ping Data for PlanetLab. [Online] http://pdos.csail.mit.edu/~strib/pl_app/.

SU, A., CHOFFNES, D., KUZMANOVIC, A., AND BUSTAMANTE, F. 2006. Drafting Behind Akamai (Travelocity-Based Detouring). In *Proc. SIGCOMM*. ACM Press, New York, NY, 435–446.

SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. 2003. Netairt: A DNS-based Redirection System for Apache. In *Proc. International Conference WWW/Internet*. IADIS, Lisboa, Portugal, 435–442.

SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. 2004. Scalable Cooperative Latency Estimation. In *Proc. 10th International Conference on Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA., 367–376.

SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. 2006a. Latency-Driven Replica Placement. *IPSJ Journal 47*, 8 (Aug.), 561–572.

SZYMANIAK, M., PIERRE, G., AND VAN STEEN, M. 2006b. Versatile Anycasting with Mobile IPv6. In *Proc. International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*. ICST, Plano, TX.

TANENBAUM, A. 2003. *Computer Networks*, 4th ed. Prentice Hall, Upper Saddle River, N.J.

TANG, L. AND CROVELLA, M. 2003. Virtual Landmarks for the Internet. In *Proc. ACM Internet Measurement Conference*. ACM Press, New York, NY, 143–152.

TANG, X. AND XU, J. 2004. On Replica Placement for QoS-Aware Content Distribution. In *Proc. 23rd INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 806–815.

TEBALDI, C. AND WEST, M. 1998. Bayesian Inference on Network Traffic Using Link Count Data. *Journal of the American Statistical Association 93,* 442, 557–576.

TERRY, D., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M., AND WELSH, B. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. 3rd International Conference on Parallel and Distributed Information Systems*. IEEE Computer Society Press, Los Alamitos, CA., 140–149.

TEWARI, R., NIRANJAN, T., AND RAMAMURTHY, S. 2002. WCDP: A Protocol for Web Cache Consistency. In *Proc. 7th International Workshop on Web Content Caching and Distribution*.

TORRES-ROJAS, F. J., AHAMAD, M., AND RAYNAL, M. 1999. Timed Consistency for Shared Distributed Objects. In *Proc. 18th Symposium on Principles of Distributed Computing*. ACM Press, New York, NY, 163–172.

VAN LANGEN, S., ZHOU, X., AND VAN MIEGHEM, P. 2004. On the Estimation of Internet Distances using Landmarks. In *Proc. International Conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking*.

VEAL, B., LI, K., AND LOWENTHAL, D. 2005. New Methods for Passive Estimation of TCP Round-Trip Times. In *Proc. Passive and Active Measurement Workshop*. Lecture Notes on Computer Science, vol. 3431. Springer-Verlag, Berlin, 121–134.

VERMA, D. C. 2002. *Content Distribution Networks: An Engineering Approach*. John Wiley, New York.

VOULGARIS, S., GAVIDIA, D., AND VAN STEEN, M. 2005. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management 13,* 2 (June), 197–217.

WALDVOGEL, M. AND RINALDI, R. 2003. Efficient Topology-Aware Overlay Network. *ACM Computer Communications Review 33,* 1 (Jan.), 101–106.

WANG, J. 1999. A Survey of Web Caching Schemes for the Internet. *ACM Computer Communications Review 29,* 5 (Oct.), 36–46.

WANG, L., PAI, V., AND PETERSON, L. 2002. The Effectiveness of Request Redirection on CDN Robustness. In *Proc. 5th Symposium on Operating System Design and Implementation*. USENIX, Berkeley, CA, 345–360.

WOLSKI, R., SPRING, N., AND HAYES, J. 1999. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems 15,* 5-6 (Oct.), 757–768.

WONG, B., SLIVKINS, A., AND SIRER, E. G. 2005. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *Proc. SIGCOMM*. ACM Press, New York, NY, 85–96.

WOOTTON, C. 1999. JavaScript Weirdness. *Web Developer's Journal*. [Online] http://www.webdevelopersjournal.com/.

XIAO, J. AND ZHANG, Y. 2001. Clustering of Web Users Using Session-Based Similarity Measures. In *Proc. International Conference on Computer Networks and Mobile Computing*. IEEE Computer Society Press, Los Alamitos, CA., 223–228.

YIN, J., ALVISI, L., DAHLIN, M., AND IYENGAR, A. 2002. Engineering Web Cache Consistency. *ACM Transactions on Internet Technology 2*, 3 (Aug.), 224–259.

YU, H. AND VAHDAT, A. 2000. Efficient Numerical Error Bounding for Replicated Network Services. In *Proc. 26th International Conference on Very Large Data Bases*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufman, San Mateo, CA., 123–133.

YU, H. AND VAHDAT, A. 2002. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems 20*, 3, 239–282.

ZARI, M., SAIEDIAN, H., AND NAEEM, M. 2001. Understanding and Reducing Web Delays. *Computer 34*, 12 (Dec.), 30–37.

ZEGURA, E. W., AMMAR, M. H., FEI, Z., AND BHATTACHARJEE, S. 2000. Application-layer Anycasting: A Server Selection Architecture and use in a Replicated Web Service. *IEEE/ACM Transactions on Networking 8*, 4 (Aug.), 455–466.

ZHANG, R., ABDELZAHER, T., AND STANKOVIC, J. 2004. Efficient TCP Connection Failover in Web Server Clusters. In *Proc. 23rd INFOCOM Conference*. IEEE Computer Society Press, Los Alamitos, CA., 1219–1228.

ZHANG, Y., ROUGHAN, M., LUND, C., AND DONOHO, D. 2003. Traffic Engineering: An Information-Theoretic Approach to Traffic Matrix Estimation. In *Proc. SIGCOMM*. ACM Press, New York, NY, 301–312.

ZHAO, B., HUANG, L., STRIBLING, J., RHEA, S., JOSEPH, A., AND KUBIATOWICZ, J. 2004. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communication 22*, 1 (Jan.), 41–53.

ZHENG, H., LUA, E. K., PIAS, M., AND GRIFFIN, T. 2005. Internet Routing Policies and Round-Trip Times. In *Proc. Passive and Active Measurement Workshop*. Lecture Notes on Computer Science, vol. 3431. Springer-Verlag, Berlin, 236–250.